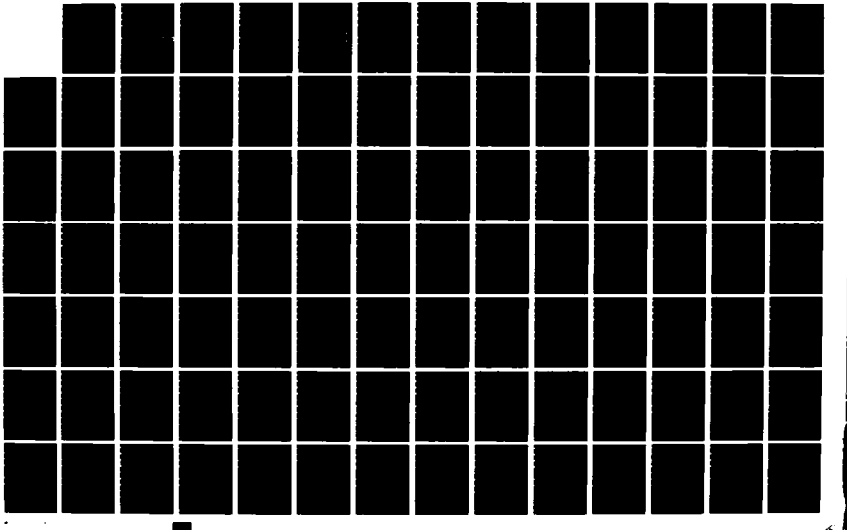
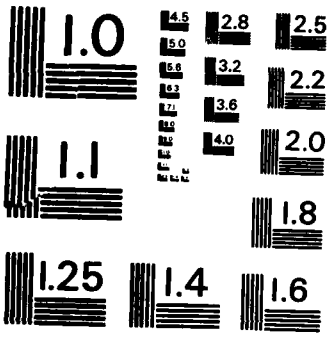
AD-A166 762 A QUANTITATIVE CHARACTERIZATION OF CONTROL FLOW CONTEXT: SOFTWARE MEASUEE.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH J W HOWATT 1985 1/2

UNCLASSIFIED AFIT/CI/NR-86-37D F/G 9/2 NL





SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 86- 370	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A Quantitative Characterization Of Control Flow Context: Software Measures For Programming Environments		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James William Howatt		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT:		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433-6583		12. REPORT DATE 1985 13. NUMBER OF PAGES 141
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. (of this report) UNCLASS 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 7 April 86
Dean for Research and
Professional Development
AFIT/NR, WPAFB OH 45433-6583

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

A quantitative characterization of control flow context: Software measures for programming environments

James William Howatt

ABSTRACT

A review of published measures of control flow complexity in programs reveals three major deficiencies: loss of information, lack of specificity, and lack of analytical support. A new approach is used to characterize the control structure of a program, with the aim of defining properties and measures of control flow that can be of immediate use to programmers, regardless of their utility as complexity measures. Mathematical rigor and analytical evaluation techniques are used to define a set of properties of control structure and a corresponding vector of measures. Instead of defining the properties and measures for an entire flowgraph, they are defined at the node level, reflecting the control flow surrounding each node in a flowgraph. The properties and their measures reflect the following characteristics of control flow: nesting, iteration, structuredness, and predecessors. Algorithms for computing the properties and their measures are presented. An assessment of the computational complexity of the algorithms shows that they are feasible programming environment tools.

A finite path set, representing all possible execution sequences, is evaluated as a characterizing property. Desired characteristics of the path set are defined and used to evaluate four published path subset criteria. Those criteria are shown to be deficient, so a fifth criterion is defined. However, the path set satisfying this fifth criterion is shown to be too large to be of practical use to a programmer.

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: _____

_____

AUTHOR: _____

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

( ) a. MAN-YEARS _____ ( ) b. $_____

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME GRADE POSITION

ORGANIZATION LOCATION

STATEMENT(s):

FOLD DOWN ON OUTSIDE - SEAL WITH TAPE

AFIT/NR
WRIGHT-PATTERSON AFB OH 45433

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 73236 WASHINGTON D.C.

POSTAGE WILL BE PAID BY ADDRESSEE

AFIT/ DAA
Wright-Patterson AFB OH 45433

FOLD IN

A quantitative characterization of control flow context:

Software measures for programming environments

by

James William Howatt

Captain, USAF

An Abstract of

A Dissertation Submitted in Partial Fulfillment

of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

141 pages

Iowa State University

Ames, Iowa

1985

A quantitative characterization of control flow context:

Software measures for programming environments

by

James William Howatt

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

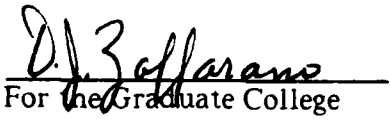
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

In Charge of Major Work

For the Major Department



For the Graduate College

Iowa State University
Ames, Iowa

1985

TABLE OF CONTENTS

I. INTRODUCTION

Software quality may currently be the paramount issue of the computing industry. The many catastrophes, mishaps, and embarrassments caused by software failure, as reported in the ACM's *Software Engineering Notes (SEN)*, underline this quality and reliability issue. Definitions of software quality and descriptions of quality assurance programs appear regularly in *SEN* and in journals such as the *IEEE Transactions on Software Engineering*. It seems that almost everyone has ideas about composing quality software. To objectively discern the quality of a software product, there must be measures of the product that accurately reflect its quality. Providing such measures is the ultimate goal of software metrics research.

Researchers in software complexity metrics investigate not only software, but also the life-cycle documents—the requirements, specifications and documentation—that support the software to identify properties that affect their quality. Their goal is to produce measures that reflect the degree of "goodness" of those products. Although most researchers agree that measuring the quality of requirements and specifications early in the software life-cycle is important, most of their current work concentrates on programs. And, because of the emphasis on using only well-defined control constructs, as in structured programming, most of the work on programs has focused on control flow. Indeed, the field of software complexity metrics seems to have grown up around the many proposed quantifications of program control structure. This dissertation presents the results of research in control structure metrics, but with a different objective than that of prior research efforts.

Before describing the goals of this research, two comments are in order about the name associated with this research area. Although it is most commonly referred to as "complexity metrics" research, both of the terms "complexity" and "metrics" are misnomers. The term "complexity" implies interaction between a programmer and a program. However, few of the proposed complexity metrics really reflect that

interaction. More often, the metrics reflect only the interaction between different program units. Since the programmer is all but left out of the picture, the connotation is undeserved. In this dissertation, the term complexity is used to refer to interaction between program control structures. No programmer-program interaction is to be inferred unless stated explicitly.

The term "metric" is also ill-chosen. According to Melton and Gustafson [1], a metric is a mathematical distance function on ordered pairs of objects. Most published "software metrics" measure only one object, the program. Thus, to avoid ambiguity, the term "measure" is used in this dissertation instead of "metric".

The goal of the research reported in this dissertation is the characterization and quantification of the control structure of programs. However, the immediate purpose is not to produce measures of programmer-program complexity. The review of previously proposed measures in Chapter II reveals that many researchers have moved too quickly in trying to provide this type of measure. Often, proposed measures are found to be too general to be useful. For example, most measures of nesting cannot reflect the difference between one program that contains a single deeply nested construct and another that contains several shallowly nested constructs. Other measures reflect far too little of the control structure. For example, predicate count measures have been touted as providing an adequate picture of control flow complexity [2, 3]; but they do not reflect properties such as nesting and structuredness. Another common deficiency is a lack of rigor and analytical evaluation. Too often, measures and the properties that they quantify are poorly defined. Further, little or no analysis is presented to show that the measures accurately reflect the property. One example of this lack of analysis is apparent in [4], in which one researcher presents a nesting measure, but generalizes the computation of that measure so much that it no longer reflects nesting. In its computed form, the measure reduces to a simple node count measure.

Finally, many measures are presented as accurate reflections of the difficulty of understanding a program. Often this claim is made with little or no empirical support. Such claims are usually based on the researcher's intuitive notion of the causes of psychological complexity. Empirical evaluations of these "complexity metrics" often produce inconclusive results. Sheil [5] argues that many empirical results are "unsatisfactory in that they are methodologically weak, the effects they report are small, and yet they are presented as if they establish claims that go far beyond their data." He goes on to explain that proper techniques must be established for evaluating complexity measures before valid, generally applicable results can be obtained. Those techniques are yet to be defined. So, instead of developing measures that are supposed to reflect the psychological complexity of programs, an alternate approach, the one reported in this dissertation, is to define properties and measures that can be of immediate use to programmers, regardless of their utility as complexity measures.

A. Problem Statement

(1) Develop a rigorous characterization of program control flow with the goal of providing information that can be of immediate use to a programmer.

(2) Define measures of the properties of the characterization.

The research focuses on imperative programs with "conventional" control flow. Concurrency, intermodular control flow, and programs coded in functional and applicative languages are not addressed.

B. Methods

The flow graph is used as the model of program control structure. A flow graph is a directed graph comprised of nodes, which represent basic blocks of program code, and arcs that represent possible control paths between basic blocks. A basic block is a sequence of program statements that can be entered only through the first statement

in the sequence and exited only through the last statement.

For a characterization of control flow to be useful to a programmer, it must be defined from the perspective of the programmer. Since a programmer usually focuses on a small part of a program at any given time, the characterization will be of most use if it reflects the control structure containing those parts. For control flow purposes, each basic block can be selected as a "part", because each statement in a given basic block is affected by control flow in the same way. Therefore, the characterization of control flow developed in this work is a characterization of the control flow surrounding each node in a program's flow graph.

Four control flow properties are considered for use in this characterization. The first of these properties is nesting. The negative effect of increased nesting depth on understandability has been commonly accepted [6, 7, 8]. Weinberg even defined a control structure that allows any construct with arbitrary levels of nesting to be expressed in a structure with just one nesting level [9]. Nesting is directly tied to the set of predicates that influence execution of a given node. The nesting property reflects all the predicators of that node.

However, nesting in an iterative construct differs from nesting in an *if* or *case* construct. In the former, a given node can affect its predicate; in the latter it cannot. The second property, one that reflects iteration containing a node of interest, provides a distinction between these types of nesting.

The third property reflects structuredness. Most programming professionals agree that "structured programming" enhances program development. But, there are advocates of deviations from structured techniques [10, 11] and those that believe that ad hoc methods suffice. Thus, since some programmers will continue to write unstructured programs, a property that reveals unstructuredness in program control flow can be a useful tool for identifying complicated control structure. The approach to characterizing structuredness is not one of "either the program is structured or

not". Instead, pairs of predicates are examined to see if their interaction violates structuring rules. This approach allows specific areas of unstructured control flow to be identified for appropriate attention.

The final property is based on a definition of a finite set of paths that represent all possible execution sequences from the start node to a given node of interest. Five desirable properties of such a path set are defined and used to evaluate four published criteria for path sets. When those criteria were found deficient, a fifth criterion was defined that possessed the desirable properties. However, the set of paths satisfying this criterion is shown to be too large to be of practical use to programmers. Instead of a set of paths, the more tractable set of predecessors of the given node is used as the fourth property to characterize control flow.

The properties and their associated measures are developed with careful mathematical rigor and are analyzed with equal care and rigor. This not only produces a well-defined characterization of program control flow, but also provides a firm analytical basis for future evaluation of the measures as complexity measures. Rigorous analytical methods are as important as good empirical techniques because, as Evangelist argues, "the weak theoretical foundation supporting the field of software metrics confounds current attempts to justify empirically the use of various metrics" [12].

C. Organization of Dissertation

Chapter II contains a review and analysis of the measures literature. Chapter III presents the development of the nesting, iteration and structuredness properties. Chapter IV discusses paths and presents the predecessor property. Algorithms to compute the measures, as well as an assessment of their computational complexity, are presented in Chapter V. Chapter V also contains examples that illustrate the utility of the properties and their measures. The conclusion, Chapter VI, summarizes the results and provides directions for future research.

II. REVIEW OF THE PERTINENT LITERATURE

This chapter contains a review and evaluation of published results in control flow measures research. Section A begins the chapter with definitions of constructs and properties of the constructs on which most control flow measures are based. Because many of the measures are not well-defined, these definitions add rigor to some of the concepts discussed. However, some of the measures are so poorly defined, even these definitions do not totally eliminate the ambiguity.

The review contained in Section B examines published control flow measures. These measures fall into the following four categories:

(1) statement, node or predicate counts,
(2) path-based measures,
(3) nesting-based measures, and
(4) structuredness measures.

Although a few measures could be assigned to more than one category, each measure usually reflects one property more strongly than others, and is categorized according to that property. Section C contains a review of measures that are not strictly control flow measures, but represent work in other areas of active measures research.

Section D contains a discussion of three deficiencies common to the majority of the reviewed control flow measures. The deficiencies motivate the problem statement given in Chapter I.

A. Definitions

This section presents definitions that provide a basis for most control flow measures. These definitions apply, except where noted, throughout the rest of this dissertation. Since control flow in programs is usually modeled by some form of directed graph, such as a flow graph or flow chart, definitions of "directed graph", of general properties of directed graphs, and of "flowgraph" begin the section.

Definition 2.1: A *directed graph* G = (N, E) consists of a set of nodes N and a set of edges E. An edge is an ordered pair of nodes (x,y). In this pair, node x is an *immediate predecessor* of node y and node y is an *immediate successor* of node x. A node z has *indegree* n if E contains exactly n arcs of the form (w,z). Node z has *outdegree* n if E contains exactly n arcs of the form (z,w).

Definition 2.2: A *path* P in a directed graph G = (N, E) is a sequence of edges $(x_1, x_2), (x_2, x_3), ..., (x_{k-2}, x_{k-1}), (x_{k-1}, x_k)$ where $\forall i [1 \leqslant i < k \Rightarrow (x_i, x_{i+1}) \in E]$. P is a path *from* x_1 *to* x_k. Each node n appearing in P *lies on* P and is denoted by $n \in P$. Similarly, each edge e appearing in P *lies on* P and is denoted by $e \in P$.

Since E is a set, for each node pair x and y there is at most one edge $(x, y) \in E$. Thus, a path in a graph can be represented unambiguously as a sequence of nodes.

Definition 2.3: A *cycle* is a path on which the endpoints coincide.

Definition 2.4: An *elementary cycle* is a cycle on which all nodes, except the end nodes, are distinct.

Definition 2.5: A *flowgraph* G = (N, E, s, t) is a directed graph with a finite, nonempty set of nodes N, a finite, nonempty set of edges E, a start node $s \in N$ and a terminal node $t \in N$. The start node s is the unique node of N with indegree zero. The terminal node t is the unique node of N with outdegree zero. Each node $x \in N$ lies on a path in G from s to t.

To maintain consistency in measures research, creating a unique flowgraph for any given program is highly desirable. One method of ensuring this uniqueness is by letting each node in a flowgraph represent a basic block in a source program.

Definition 2.6: A *basic block* is a (longest) sequence of code that is entered only at the beginning of the sequence, is executed sequentially, and is exited only at the end of the sequence.

The nodes s and t do not represent basic blocks. They are used only as markers in a flowgraph to represent starting and ending points. Representing basic blocks as nodes not only provides a uniform method for constructing flowgraphs, but also conforms to the flowgraph construction techniques used in data flow analysis [13], providing consistency across applications.

Probably the most important type of node, from a control flow viewpoint, is the decision, or predicate, node. These nodes represent those places in a program where choices must be made as to the particular path to be followed.

Definition 2.7: In a flowgraph G = (N, E, s, t), a *predicate, or decision, node* is any node in N with outdegree greater than one. A *binary* predicate node has outdegree of exactly two.

Decision nodes create two basic control constructs: alternation and iteration. Alternation, usually represented by an *if* statement, is the splitting of one path into two or more paths that eventually rejoin at some point "further down" in the program. Iteration is also the splitting of one path into several (usually two) such that one of the paths branches "back up" into the program forming a cycle. Associating cycles with the nodes that predicate them plays an important role in determining the paths in a flowgraph. Intuitively, a decision node predicates a cycle if it has one outarc that lies on the cycle and another that does not. This notion is given formally in the following definition.

Definition 2.8: A node p *predicates a cycle C* if $p \in C$ and there exists a path from an immediate successor of p to t that contains no nodes in C.

While cycles are important for path analysis, programmers usually think of iteration more in terms of loops than cycles. A loop is the collection of cycles predicated by p.

Definition 2.9: The *loop predicated by node p* is the set of all cycles predicated by p.

Although this definition suffices for the measures discussed in Section B, it is not restrictive enough for a truly rigorous definition of iteration. A more restrictive definition is given in Chapter IV, where it is needed to define the concept of a representative execution path. The meaning of execution path is given by Definition 2.10.

Definition 2.10: For a flowgraph G = (N, E, s, t), any path P from s to t is a *(possible) execution path.*

The ordering of nodes along an execution path is another important property. For some of the reviewed measures and, more importantly, for the measures developed in Chapter III, it is important to be able to determine which nodes *always* precede others along all execution paths. This is the concept of dominance [13]. If the first occurrence of node x precedes the first occurrence of node y on all possible execution paths, then node x is said to dominate node y.

Definition 2.11: If p and q are any two nodes in a flowgraph G = (N, E, s, t), then p *dominates* q if p lies on every path from s to q. Node p *properly dominates* node q if p dominates q and $p \neq q$.

Sometimes it is important to know not the entire set of dominators for a given node x, but only the closest dominator, the immediate dominator of x.

Definition 2.12: Let p, q and r be nodes in flowgraph G. Node p is the *immediate dominator* of q if p properly dominates q, and if r properly dominates q and $r \neq p$, then r properly dominates p.

An important property of immediate dominators, proved by Hecht [13], is that for any node $x \in N - \{s\}$, the immediate dominator of x is unique. (The start node has no predecessors, and, therefore, no immediate dominator.)

Along with paths, loops, and their associated properties, structured programs and structured constructs are often used as a basis for measures and for assessing the

utility of measures. Structured flowgraphs are those flowgraphs built from the sequencing and nesting of elementary structured constructs.

Definition 2.13: An *elementary structured construct* is any of the constructs shown in Figure II-1.

Although the n-½ loop is not usually regarded as an elementary structured construct, it possesses the single-entry, single-predicate property of the others and is, therefore, included here.

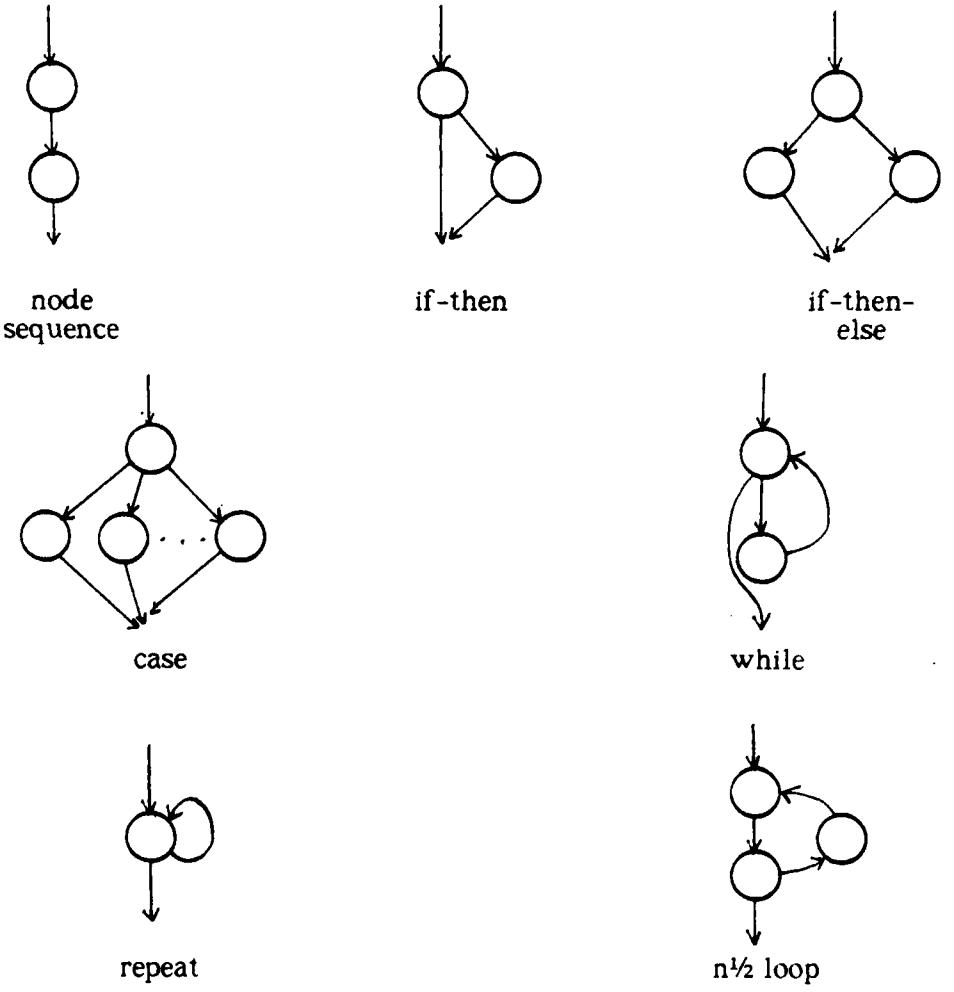


Figure II-1. Elementary structured constructs

B. Review of the Measures

1. Simple counting measures

The measures in this class are those based on simple counts of elementary program properties. Perhaps the most widely used measure in this category is the count of lines of code in a program. However, as Harrison et al. [14] point out, this simple measure is not well-defined. It has been used to mean (1) number of program statements, (2) total source lines, (3) total executable source lines, and (4) total declaration and executable source lines. Although the measures are much maligned, the fourth interpretation was found by Sheppard and her colleagues to perform as well as or better than more sophisticated measures in experiments predicting program comprehensibility [15]. Schneidewind and Hoffman also found statement count to correlate well with error occurrences [16].

Statement counts are often considered general complexity measures. To focus on control flow, some researchers propose counting only those statements that affect the flow of control in a program. Gilb [3] defined his *absolute logical complexity* as the number of binary predicates in a program. But, he felt that this measure was inadequate for comparing two programs. He reasoned that one program containing 10 predicates and 10 statements had relatively more involved control flow than another program containing 10 predicates and 100 statements. To reflect this difference, he proposed a second measure, the *relative logical complexity*, as the absolute logical complexity divided by the number of statements in the program.

Feuer and Fowlkes [8] proposed a relative measure based solely on a flowgraph: the number of predicate nodes divided by the total number of nodes. They also proposed using the cardinality of the flowgraph's node set (and other measures discussed in subsequent sections) in an attempt to identify measurable properties of programs that influence maintainability. They reasoned that maintenance performance depends on the complexity of the algorithm coded and on the clarity of

the coding. They wanted their measures to reflect these properties, so they required the measures to be language independent and noncoercible. Language independence means that a measure can consistently be applied to programs written in several programming languages and that the ordering the measure assigns to a set of algorithms remains reasonably constant when the algorithms are coded in languages of comparable power. A measure is noncoercible if it accurately reflects the property of interest. An example of a coercible measure is using a count of comment lines in a program to assess the quality of the documentation. One can add or delete comments without affecting the documentation quality.

To test if their measures reflected program maintainability, Feuer and Fowlkes applied them to 123 PL/I program modules for which maintenance records had been kept for one year. From the records, they extracted two values that they thought characterized maintenance performance: the total time spent maintaining the module and the number of changes made to the module. The maintenance manhours were obtained from "informal and incomplete" time records; the change data were obtained from records kept by an automated maintenance system.

Their analysis was incomplete when they wrote their paper, but some results were evident. The time to repair errors was highly correlated with node count, but not with the ratio of predicate nodes to total nodes. They concluded that node count appears to be a good indicator of maintenance performance, but that more research and analysis is necessary.

Myers [17] also thought that predicate counts could be used to quantify complexity, but that the count alone was not sufficient. As an example he presented the following three statements:

```
(A)  IF (X = 0) THEN ...
          ELSE ...

(B)  IF (X = 0) & (Y > 1) THEN ...
                ELSE ...
```

```
(C)  IF (X = 0) THEN
       IF (Y > 1) THEN ...
             ELSE ...
```

He argued that statement A is intuitively less complex than statement B because of the extra condition in statement B. He also said that statement B is less complex than statement C because statement C contains two predicates, one nested within the other. However, a predicate count measure ranks the first two the same. To obtain a more reflective ordering, Myers proposed that complexity be represented by an interval with one plus the number of predicates as the lower bound, and one plus the number of simple conditions as the upper bound. Applying his measure to statements A, B and C yields 2:2, 2:3 and 3:3, respectively, values that satisfy his intuitive ordering.

Hansen argued that Myers' measure ignores sources of complexity other than control flow [18]. He asserted that any complexity difference between statements A and B is not caused by control flow; both statements produce the same flowgraph. Instead, the difference is in the complexity of their conditions. To account for this, he proposed measuring complexity by the pair (predicate count plus 1, operator count). The operator count reflects complexity from sources other than control flow. He defined the operators to be:

primitive operators (+, -, *, "and", "or", etc.)
assignment operator
subroutine and function calls
array subscripts
input/output statements

Hansen claimed that his measure is a better reflection of the overall complexity than Myers' because the components of the pair are independent. Applying his measure to statements A, B and C in Myers' example yields (2,1), (2,3), and (3,2), respectively. This measure does not necessarily reflect Myers' ordering because, as Baker and Zweben [19] point out, it fails as a tool for comparing two programs. If the measure for a program P_1 is (u,v) and for a program P_2 is (y,z), and if $u < y$ and

$v > z$, nothing can be inferred about the complexity difference of the two programs. The measures for statements B and C also illustrate that an implementation reducing one type of complexity can increase another type. In statement B, the lower control flow complexity results in higher expression complexity, while in statement C the lower expression complexity is offset by the higher control flow complexity.

Although some of the counting measures described in this section correlate well with experimental data, they reflect very little of the structure of the program. It seems reasonable that the order in which these nodes can be executed, where they fall on an execution path, affects maintainability and error occurrences more than simply their existence.

2. Path-based measures

One of the first path-based measures was defined by McCabe [2]. To enhance the testability and maintainability of programs, McCabe proposed limiting module sizes by limiting the number of paths through a module. However, counting all paths is time consuming, and complicated by the possibly infinite number of paths caused by cycles. So instead of counting total paths, McCabe chose to count a module's basic paths. Basic paths are those that when taken in combination can produce all other paths in the module. These paths are linearly independent; each contains an arc not found in any of the other basic paths.

To compute the number of basic paths, the size of the basis, McCabe borrowed from graph theory. He chose the *cyclomatic number* of a graph, the maximum number of linearly independent cycles in the graph. For a graph $G = (N, E)$, the cyclomatic number is given by

$$V(G) = |E| - |N| + p$$

where p is the number of connected components in G. The formula applies only to fully connected graphs, graphs in which there is a path from every node to every other node in a connected component. McCabe found that a set of basic paths could

be obtained from a set of basic cycles, and that the cardinalities of the two sets were equal. To apply the cyclomatic number formula to a flowgraph, McCabe had to make the flowgraph fully connected by adding an arc from t to s, increasing the number of arcs by one. By definition, a flowgraph consists of just one connected component; therefore, p = 1. Thus, the number of basic paths through a flowgraph G = (N, E, s, t), its *cyclomatic complexity*, is given by

$$V(G) = |E| - |N| + 2.$$

McCabe provided no procedure for generating either a cycle basis or a path basis. However, a procedure for generating the latter, due to Baker [20], is given in Section IV.B.2.

To see how the cyclomatic complexity is computed, consider the flowgraph in Figure II-2, taken from McCabe's paper. The graph has 11 arcs and 8 nodes, giving it a cyclomatic complexity of 11 - 8 + 2 = 5. One set of basic cycles is {sabefts, beb, abea, sacfts, sadcfts}. The cycle sabeabebeacfts is a combination of the cycles abea,

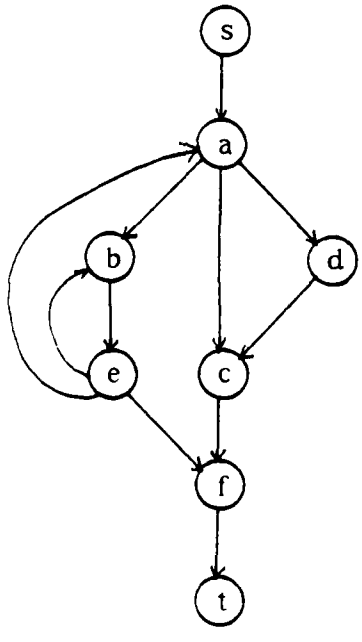


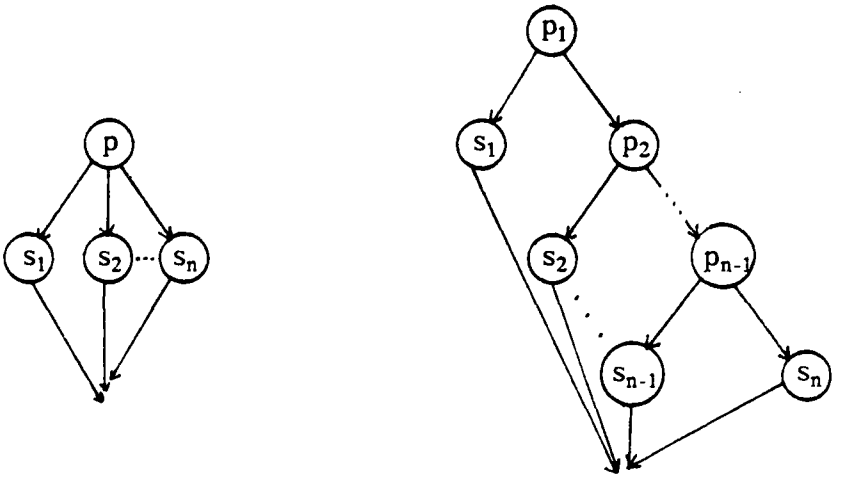
Figure II-2. McCabe's sample flowgraph

beb (twice), and sacfts. A set of basic paths derivable from the cycle basis is {sacft, sadcft, sabeft, sabebeft, sabeabeft}. The path sabeabebeacft is a combination of the basic paths sacft, sabebeft, and sabeabeft.

McCabe showed that the cyclomatic complexity of a structured program is one more than the number of its binary predicate nodes, assuming that a node with outdegree n represents n-1 binary predicates. This observation allows the cyclomatic complexity of a program to be computed directly from its source code, alleviating the need to construct a flowgraph. He also asserted that this holds for unstructured programs, but provided no proof.

The proof is easily constructed. First, it must be shown that the cyclomatic complexity of an n-way branching construct is equal to the cyclomatic complexity of an equivalent nested *if* construct. Consider the flowgraphs in Figure II-3 of such constructs.

Lemma 2.1: The cyclomatic complexities of flowgraphs G_a and G_b in Figures II-3(a) and II-3(b), respectively, are equal.



(a) (b)

Figure II-3. Equivalent n-condition constructs

Proof: In flowgraph G_a, $|N| = n+1$, $|E| = 2n$, and $V(G_a) = 2n - (n+1) + 2 = n+1$. In flowgraph G_b, $|N| = n-1 + n$, $|E| = 2(n-1) + n$, and $V(G_b) = 2(n-1) + n - (n-1 + n) + 2 = n+1$. □

By the proof of the lemma, any flowgraph G with cyclomatic complexity c that contains n-branch predicates, $n > 2$, can be represented by a flowgraph G′ that contains only binary predicates having the same cyclomatic complexity. Now it must be shown that if a flowgraph contains only binary predicates, then its cyclomatic complexity equals one plus the number of predicate nodes.

Theorem 2.1: Let $G = (N, E, s, t)$ be a flowgraph. Let $P \subset N$ be the set of predicate nodes in G. If every node $p \in P$ has outdegree 2, then $V(G) = |P| + 1$.

Proof: Since every node but t has an outarc, and every $p \in P$ has one additional outarc, $|E| = |N| - 1 + |P|$. Thus,

$$V(G) = |E| - |N| + 2$$
$$= |N| - 1 + |P| - |N| + 2$$
$$= |P| + 1 \quad \square$$

Basili [21] criticized McCabe's measure, arguing that an n-way *case* statement, treated as n-1 nested predicates by McCabe, is easier to understand than the n-1 predicates because of its symmetry. He proposed that n-way *case* statements should contribute $\log_2(n)$ to the cyclomatic complexity instead of n-1.

Schneidewind and Hoffman evaluated cyclomatic complexity and other path-based measures in an experiment to see if program structure has a significant effect on error occurrences and their detection and correction [16]. For their experiment, one programmer wrote 102 procedures for 4 small systems. One system involved string processing, two manipulated graphs, and one was a data base application. Throughout the project the programmer recorded the number and types of errors found, the time required to find them, and the time required to fix them. The researchers compared this data to the following measures: cyclomatic complexity,

number of source statements, number of paths, and reachability, which is the sum over all the nodes of the number of paths from the start node to each node. They did not rigorously define "path"; they stated only that their definition excludes paths which contain loops traversed two or more times in succession.

Of the 102 procedures, 31 contained errors other than syntax errors. These procedures were placed in one group, and the remaining 71 were placed in a second group. The researchers computed each measure for all the programs in both groups. The average value of each measure for each group is shown in Table II-1. The error group consistently produced higher average complexity values than the correct group.

Further analysis revealed that 64 errors specifically concerned program control structure. Over 62 per cent of these errors occurred in modules with a cyclomatic complexity greater than 4, and 82 per cent occurred in modules containing 5 or more paths. The average time to locate errors was 50 per cent higher in modules with a cyclomatic complexity greater than 5 than in those with V(G) less than or equal to 5. These results lend weight to the argument that increased numbers of paths increase complexity. Interestingly, no one of their measures was found to be significantly better than any of the rest for predicting error occurrences.

The authors concluded that although the correlation results were not strong enough to support the use of the measures as prediction tools, they appeared to show a connection between the occurrence of errors and program structure. Schneidewind and Hoffman support using the measures as indicators of errors, but not as absolute

Table II-1. Average values of complexity measures

Measure	Procedures	
	without errors	with errors
Cyclomatic Complexity	1.7	4.7
Number of Statements	9.4	27.2
Number of Paths	2.7	27.1
Reachability	10.1	120.3

predictors. They feel that more experiments on larger projects are needed.

Feuer and Fowlkes [8] investigated the utility of path count and average path length as measures in their experiment with node and predicate counts. Their paths were "simple paths", paths containing no more than one iteration of each loop. The notion of "loop" was not defined. The experimental results showed little correlation between these two measures and error occurrences. This may have resulted in part because they estimated the number of paths and their lengths instead of computing them outright.

The basic problem in developing path-based measures is choosing a finite set of execution paths that adequately represents the set of all execution paths. This requires an adequate, rigorous definition of loop that is also intuitively satisfying. In the reviewed path-based measures, this has not been done. In Chapter IV, definitions are developed for both "loop" and a finite set of paths useful in measures research.

3. Nesting measures

The measures reviewed thus far have been based on node, predicate, and path counts. None have attempted to capture complexity caused by statement nesting. That nesting is a source of complexity has been pointed out by Weissman [7] in his review of factors that should be considered as complexity contributors, and by Weinberg et al. [9] who suggested developing programming constructs that eliminate the need for nesting.

Dunsmore and Gannon were among the first to experimentally evaluate a nesting complexity measure [22]. In their experiment, they had 31 college seniors and graduate students write programs that would input a string, use a recursive procedure to reverse it, and then print the reversed string the number of times specified in the input. The subjects were divided into two groups, each group using a different variation of the SIMPL [23] programming language. Both versions contained identical control constructs; they differed only in data typing capabilities. As the subjects

worked on their programs, they recorded the number of errors found and number of changes made.

For each completed program, the researchers computed the average nesting depth as the sum of the nesting depths of all the statements divided by the number of statements in the program. To determine nesting levels, they assigned unnested statements to level 1 and specified that if a *case*, *while*, or *if* statement resided at level i, then the statements embedded within those constructs resided at level i+1. Since only structured constructs were allowed, the levels were easily computed.

Dunsmore and Gannon hypothesized that programs with more nesting would produce more errors and require more changes than programs with little or no nesting. The results were opposite of what they expected. Programs with fewer errors and changes had greater average nesting depths. Although the results were not statistically significant, Dunsmore and Gannon surmised that the better programmers better understood the problem and took advantage of what they called "the abstraction capability provided by nesting."

A second nesting measure was proposed by Chen [4] in an attempt to derive a functional relationship between programmer productivity, as measured by lines of code produced per unit time, and program complexity, as measured by his control structure entropy measure. The measure, adapted from an information-theoretic entropy formula, reflects control structure by counting the new nesting levels in a program. A new level of nesting is created by a predicate when it does not sequentially follow another predicate. A predicate p sequentially follows a predicate q if both are nested in exactly the same constructs and all paths from q to the exit node contain p. Relative nesting was easily determined because Chen worked with programs composed of only *if-then*, *if-then-else*, and *while* control flow operators.

To compute Chen's measure for a flowgraph, the predicates in that flowgraph must be ordered. The ordering is arbitrary, except that the first predicate in the

program must be first in the ordering. Then, for a program containing n predicates, the measure is computed as

$$Z_n = 1 + \sum_{i=2}^{n} \log_2(2p_i+q_i)$$

where q_i is the probability that the ith predicate sequentially follows any of its predecessor predicates, and p_i is equal to $1-q_i$. In any given flowgraph, the probabilities can only be 1 and 0. As an example, consider the flowgraph in Figure II-4. Predicates 2 and 3 do not sequentially follow any other predicates; therefore, $q_2 = q_3 = 0$. But, predicate 4 sequentially follows predicate 3, and predicate 5 sequentially follows predicate 1; therefore, $q_4 = q_5 = 1$. Thus, the measure for the flowgraph is given by:

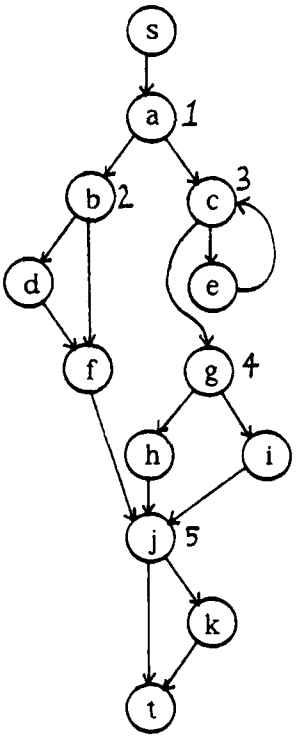


Figure II-4. Flowgraph for Chen's entropy measure

$$Z_5 = 1 + \log_2(2(1)+0) + \log_2(2(1)+0)$$
$$+ \log_2(2(0)+1) + \log_2(2(0)+1)$$
$$= 3$$

To simplify the measure's computation, Chen proposed that for an arbitrary program it is reasonable to assume that the chance of one predicate sequentially following another is ½. Thus, he generalized his measure so it depends only on the number of predicates in a program. He substituted ½ for q_i and p_i, obtaining the formula

$$Z_n = 1 + \sum_{i=2}^{n} \log_2(2(½)+½)$$

$$= 1 + (n-1)(\log_2 3 - \log_2 2)$$

$$= (n-1)\log_2 3 - n + 2$$

To validate this general measure, Chen had programmers who were working on eight different programming projects record the time they spent on each project. The times included all work from studying the specifications to preparing the final documentation. From these records, Chen computed the productivity of each programmer. For each of the programs, he computed the general entropy measure. When he analyzed the results, he found, as expected, that programmer productivity was lower for the programs with higher complexity measures.

Although these results seem reasonable, one questions the reasoning behind generating the general formula. Because the measure depends only on the number of predicates, it cannot reflect their structure—the measure no longer reflects nesting.

A third nesting measure was defined by Harrison and Magel [6, 24], who felt that McCabe's cyclomatic complexity reflects only part of the total control flow complexity in a program. In particular, it ignores complexity caused by statement nesting. To define a measure that did reflect nesting, they first developed a general

definition of nesting.

They started by borrowing the notion of lower bounds from lattice theory. While acknowledging that, in general, flowgraphs are not lattices, they argue that the idea of a lower bound still applies. For a flowgraph G = (N, E, s, t), they define the lower bounds of a node x to be the set of nodes that lie on every path from the immediate successors of x to t. The greatest lower bound of x, GLB(x), is the lower bound that precedes all the other lower bounds. For a nonpredicate node x, GLB(x) is x's immediate successor. For a predicate node p, GLB(p) is the first node following p whose execution does not depend on the truth value of p. That is, if p is executed, then GLB(p) will also be executed (barring any errors that halt the program prematurely). The execution of any of the nodes that lie on the paths from the immediate successors of p to GLB(p) depends on the truth value of p. This set of nodes, called the *scope* of p, is nested within the control influence of p. A node x is nested within a predicate p only if x falls within the scope of p. Table II-2 shows the greatest lower bound and scope for each predicate node in the flowgraph of Figure II-4.

Harrison and Magel based their measure on the scope definition. They first assumed that associated with each node in a flowgraph is its "raw" complexity, the intrinsic complexity of the code represented by the node. They suggested that this could be computed using Halstead's effort measure [25]. To capture the complexity caused by nesting, they assumed that the complexity of a predicate node depends

Table II-2. Greatest lower bounds and scopes for the flowgraph of Figure II-4

Node	GLB	Scope	Scope Number
a	j	{b,c,d,e,f,g,h,i}	9
b	f	{d}	2
c	g	{c,e}	3
g	j	{h,i}	3
j	t	{k}	2

directly on the aggregate raw complexities of the nodes in its scope. Therefore, for each predicate node they computed an "adjusted" complexity as the sum of the raw complexities of the nodes in the scope of the predicate plus the predicate's own raw complexity. (The adjusted complexity of a nonpredicate node is defined to be its raw complexity, and the adjusted complexities of s and t are zero.) They then defined the complexity of the program, the program's *scope number*, as the sum of the adjusted complexities of all the nodes in the flowgraph except the start and exit nodes. The last column of Table II-2 shows the adjusted complexity for each predicate node. Since the source code is not available, the raw complexity of each node is assumed to be 1. The scope number for the flowgraph of Figure II-4 is 25.

To evaluate their measure, Harrison and Magel computed scope number for each of the example flowgraphs presented in McCabe's paper [2]. Here again, all raw complexity values were assumed to be 1. When they ranked the flowgraphs by scope number and by cyclomatic complexity, they found the rankings disagreed for those programs containing many levels of nesting. The scope number reflected nesting complexity not captured by cyclomatic complexity. Further, the scope number provided a finer ordering than McCabe's measure, which produced the same complexity value for several of the flowgraphs. These results support the contention that scope number provides a more complete reflection of control flow complexity than does McCabe's cyclomatic number, but do not support the claim that scope number is a good general complexity measure.

Piwowarski used both scope number and cyclomatic number as the basis for his measure [26]. He felt that while capturing nesting complexity is necessary for reflecting total complexity, it is not enough. He wanted a measure that would rate (1) structured programs less complex than unstructured ones, (2) nested constructs more complex than sequential ones, and (3) n-way *case* statements less complex than n-1 nested *if* statements. To accomplish this, he defined his complexity measure N as

$$N = V^*(G) + \sum_{n \in N} P(n).$$

$V^*(G)$ is a variant of McCabe's $V(G)$ in which n-way *case* predicates count as just one predicate. $P(n)$ is the number of predicate scopes that are contained in or overlap with the scope of n. The scopes of predicates A and B overlap if neither predicate falls in the other's scope, but their scopes have at least one node in common. In the flowgraph of Figure II-5a, the scope of node x overlaps the scope of node y because node z falls within the scope of both.

Applying Piwowarski's measure to this flowgraph yields

$$N = V^* + P(w) + P(x) + P(y)$$
$$= 4 + 2 + 1 + 1$$
$$= 8$$

When this flowgraph is structured by splitting node z and placing one copy in the scopes of x and y, as shown in Figure II-5b, the overlap is eliminated and the

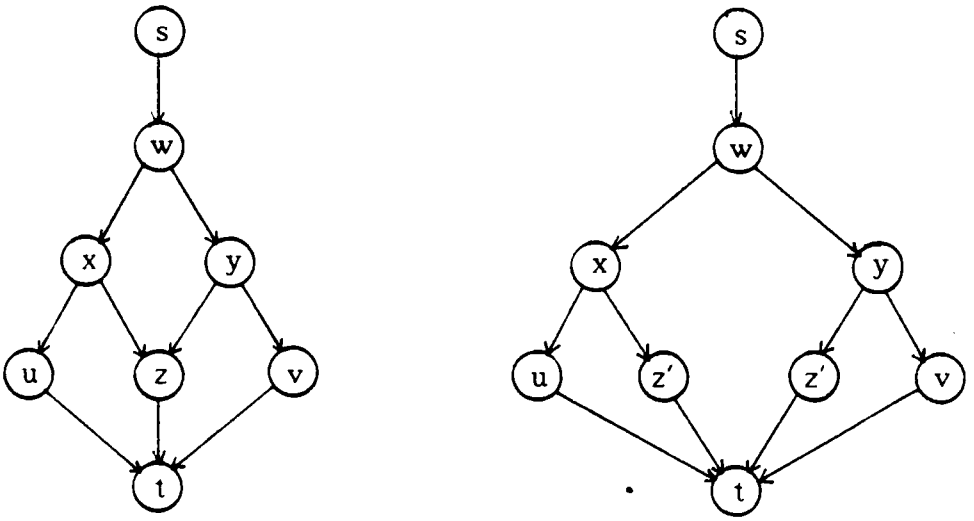


Figure II-5. Flowgraph with overlapping scopes before and after structuring

measure's value is reduced to N = 4 + 2 + 0 + 0 = 6.

Although this measure appears to be an improvement over both cyclomatic number and scope number, its ranking of an *if-then* the same as a 10-way *case* statement is intuitively unappealing. Piwowarski appears to have gone to the opposite extreme from McCabe on *case* statements.

4. Structuredness measures

For all the emphasis placed on structured programming in the past 15 years, few measures of structuredness have been proposed. Those most often described are based on repeatedly reducing a flowgraph by replacing each elementary structured construct with a single node until no such constructs remain, as illustrated in Figure II-6. If the resulting flowgraph is a single node, then the original flowgraph was fully structured. If not, then supposedly only the unstructured constructs remain. However, this method does not work if a series of nested structured constructs contains an unstructured construct at the deepest nesting level, as shown in Figure II-7. Although the code is "mostly" structured, no reductions can be done because no elementary structured constructs can be isolated.

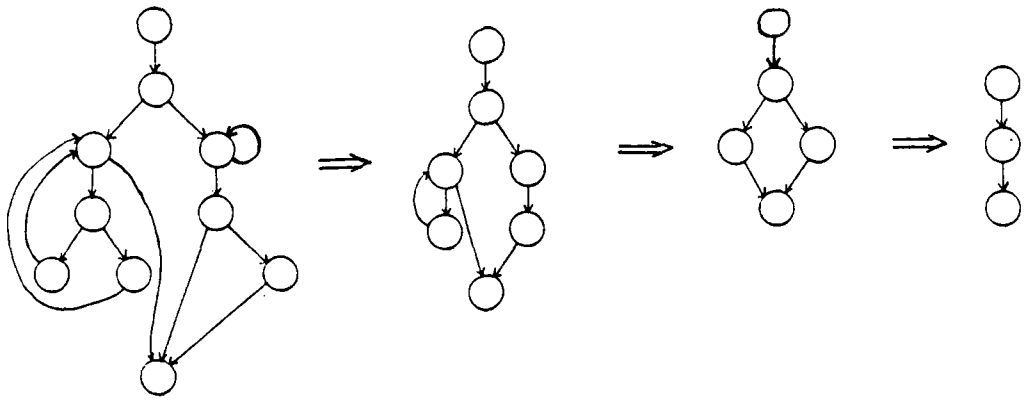


Figure II-6. Flowgraph reduction by structured constructs

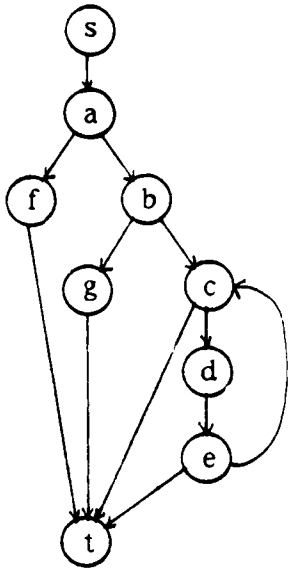


Figure II-7. A program not reducible by structured constructs

McCabe [2] used this reduction technique as the basis for his measure, *essential complexity*. He first reduces a flowgraph as described above, and then computes the essential complexity, ec = |N| - (number of reduced constructs). Thus, the essential complexity of the flowgraph in Figure II-6 is ec = 10 - 5 = 5.

The reduction technique defined by Feuer and Fowlkes [8] produces the same results as the one described above, but is defined as follows:

```
Repeat
  Replace all nodes with indegree = outdegree = 1 with an arc;
  Remove all redundant arcs and self-loops
Until no further reductions can be made.
```

This method is demonstrated in Figure II-8. However, since this method is essentially similar to the reduction by structured constructs, it will also not reduce the flowgraph of Figure II-7.

The measure based on Feuer and Fowlkes' reduction procedure, called *per cent reduction*, is the ratio of the nodes in the reduced flowgraph to the number of nodes in the original. The smaller the ratio the better structured the flowgraph. This measure

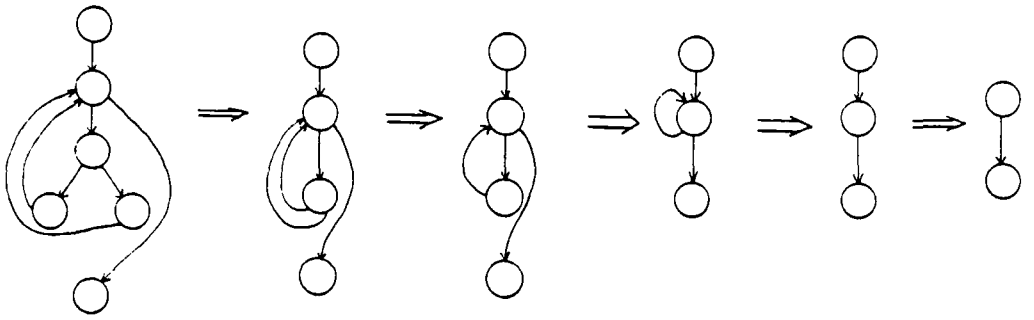


Figure II-8. Flowgraph reduction by Feuer and Fowlkes' method

did not correlate well with the maintenance data in their evaluation described in Section B.1.

Feuer and Fowlkes' reduction method is not the same as the T1 and T2 reduction method used in data flow analysis [13]. That method is defined as:

```
Repeat
  Delete all self loops;
  For each node x with only one immediate predecessor y do
    For each immediate successor z of x do
      Delete arc (x,z);
      Add arc (y,z)
    endfor
    Delete node x;
    Delete arc (y,x)
  endfor
Until no further reductions can be made.
```

This reduction technique successfully reduces all constructs except multiple-entry loops. Thus, it would only partially reflect unstructuredness in a flowgraph.

Woodward and his colleagues [27] developed a measure based on a program's source code. They compute the measure by drawing, in the left margin of a program listing, lines from each statement containing an explicit transfer of control to the destination statement(s). After all the lines have been drawn, any crossing of two lines creates a *knot*. The number of knots is the measure of complexity. Figure II-9

contains a program segment for which the knots have been identified.

Because the method of drawing lines is not well-defined, Woodward derived a more rigorous definition. Let the ordered pair (a,b) represent a branch from line a to line b, where a and b represent line numbers. Branch (p,q) causes a knot with respect to branch (a,b) if

(1) $\min(a,b) < \min(p,q) < \max(a,b)$
and $\max(p,q) > \max(a,b)$
or
(2) $\min(a,b) < \max(p,q) < \max(a,b)$
and $\min(p,q) < \min(a,b)$.

The knots measure does not reflect the structure of a program's control flow as much as it reflects the structure of the source code. In one example from their paper, they showed how rearranging the source code in a program segment reduced the number of knots. But, the flowgraph for the code with the fewer knots was unstructured while the flowgraph for the original code was structured. Therefore, the knots measure is not a true "structuredness measure".

McCabe, in addition to proposing his essential complexity measure, described the basic causes for unstructuredness in programs. He and M. H. Williams [28] independently identified the four basic unstructured constructs, at least one of which must occur in any unstructured program. These constructs are:

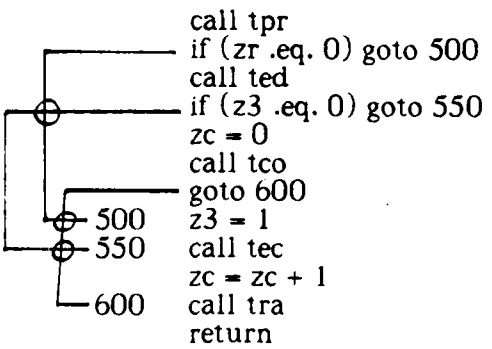


Figure II-9. Program segment with knots identified

(a) an abnormal selection path, or branch out of a decision,
(b) a loop with multiple exit points,
(c) a loop with multiple entry points, and
(d) overlapping loops.

Flowgraphs corresponding to these constructs are shown in Figure II-10. Williams included a fifth construct, parallel loops, shown in Figure II-11. However, this construct is just two multiple-exit loops connected at predicate node a.

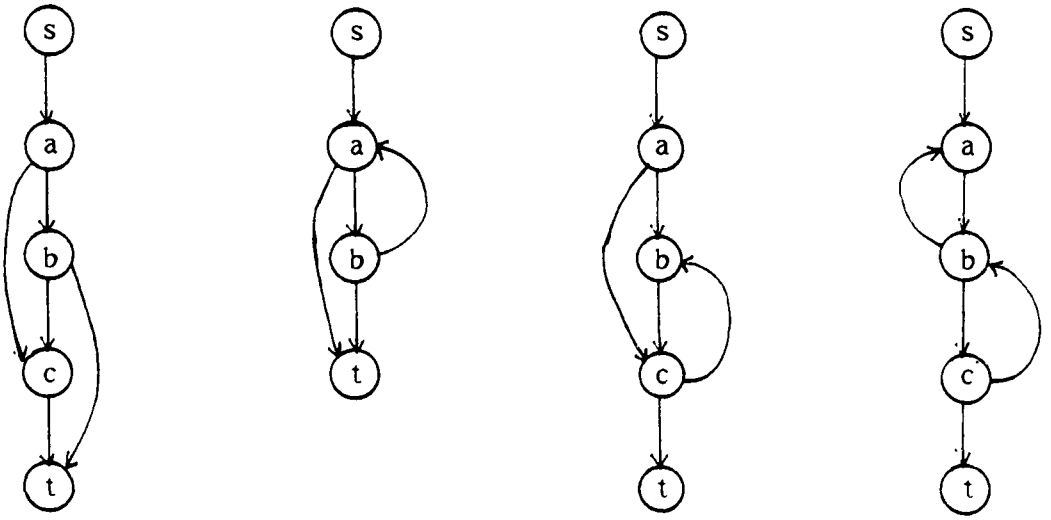


Figure II-10. Basic unstructured constructs

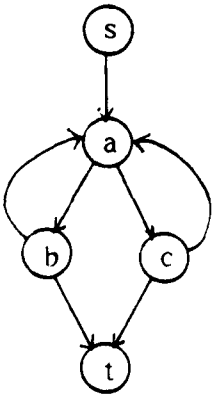


Figure II-11. William's 5th unstructured construct

From these constructs, McCabe observed that unstructuredness requires the interaction of at least two predicates. No predicate can create an unstructured construct by itself. However, McCabe failed to use this observation in his structuredness measure. The observation is used for the basis of a structuredness definition developed in Chapter III.

C. Non-Control Flow Measures

This section surveys proposed measures of complexity caused by factors other than control flow. Measures of only control flow cannot reflect all of the complexity contained in a program. Although control flow measures have received the most attention, researchers have been investigating other contributions to complexity. There appear to be three major complexity research areas besides control flow: software science, data flow/dependency, and program format. Although software science is fairly old, it is still widely used and investigated. That it was one of the first attempts to quantify program complexity makes it worth reviewing.

Data flow/dependency measures have been developed more slowly than control flow measures. Unlike the flowgraph for control flow, there has been no obvious method for modeling data flow. However, models such as the data dependency graph [29] have recently been developed and are being used in analyses of data dependency and flow in programs.

Measures of program format deal with the structure and layout of the source code itself. They address such properties as indentation, variable naming, documentation and spacing. This category of measures has received much less attention than the others and relies heavily on psychological studies instead of analytical evaluation. Since almost every programmer has his or her own personal idea of ideal formatting and variable naming, there have been few major findings in this area. While this category is worth mentioning, there is little to report about it. Therefore, the following sections will review measures in only the software science

and data dependency areas.

1. Software science measures

Maurice Halstead began developing the field of software science in the early 1970's [25]. Because of his work with assembly languages, he viewed programs as sequences of operators and operands. His aim was to develop predictions of programming effort and time as functions of operator and operand counts. To develop those functions, he first defined the following basic measures:

n_1 = number of unique operators
n_2 = number of unique operands
N_1 = total operator occurrences
N_2 = total operand occurrences

Operator and operand counts are usually made for one program module such as a subroutine, procedure, or function, but have been computed for entire programs and for parts of modules.

From these basic measures, the size of the module's vocabulary can be computed as

$$n = n_1 + n_2$$

and the length of the module, in terms of the number of tokens in the executable portion of the code, is computed as

$$N = N_1 + N_2.$$

Halstead developed additional measures in terms of these. He defined the *program volume* as the number of bits needed to encode the program in binary. The volume is computed as

$$V = N\log_2 n.$$

Since an algorithm can be realized in code in many different ways, Halstead took the view that the smallest implementation was the best. The minimal size implementation of an an algorithm is said to have volume V^*, its *potential volume*. The *program level* is the ratio of the potential volume to the actual implementation

volume. It is given by

$$L = \frac{V^*}{V}.$$

A program with L = 1 is the minimal volume implementation. The *difficulty* of a program is the inverse of its level:

$$D = \frac{1}{L}.$$

The *effort* required to write a program is computed as

$$E = \frac{V}{L} = DV.$$

As the volume or difficulty increases, so does the effort required for implementation. According to Halstead, E is the number of elementary mental discriminations needed to implement the module. Since V^* is a theoretical value, and cannot be computed, an estimator for L is needed instead. Gordon [30] proposed the approximation

$$L' = \frac{2}{n_1} \cdot \frac{n_2}{N_2}.$$

The first term is the ratio of the minimal number of operators necessary to invoke a function (the function name and the parameter grouping operators) to the actual number of operators used in the module. The second term is the ratio of the expected operand uses (each theoretically should be used just once) to the total actual operand occurrences. Thus, the more operators used and the more uses of operands, the lower the level of the implementation and the higher its difficulty. This estimate for level is substituted for L in the effort equation to obtain an effort estimation.

Although Halstead produced much empirical evidence to support software science, it has received much criticism. First, the method of determining operators and operands was never well-defined. Lassez et al. [31] argued that in some languages deciding whether a token is an operator or an operand is impossible. He gave as an example the use of a function identifier both as an invocation of the function and as a parameter to another procedure in a parameter list. Halstead ruled out this dual use

of identifiers. Secondly, Halstead omitted all program declaration statements from his counting strategy. Elshoff [32] showed that it is not reasonable to ignore them. particularly in COBOL programs where the declarations represent a significant pa » the code.

Shen and his colleagues [33] criticized the empirical evidence offered by Halstead for four reasons:

(1) The sample sizes in most of his experiments were too small to make valid statistical inferences.

(2) The programs involved were small; the results could not necessarily be generalized to larger programs.

(3) Many experiments involved only a single subject, and often the subject was also the experimenter.

(4) In experiments with several subjects, the subjects were usually college students, making a generalization to professional programmers questionable.

Shen does point out, however, that results of more recent experiments evaluating software science measures tend to support the utility of the measures.

Software science measures must be considered general measures of program complexity. That they cannot reflect just control flow in programs is shown in the following example. Consider the following code segments:

(A) x := x + f(a,0) + 12 * y

(B) if x = 0 then
      if y > 12 then
        a := f(y)

Table II-3 lists operators and operands and gives the basic measures for each code segment. Since the corresponding measures for each segment are equal, all derived measures for each will also be equal. Thus, the difference in the control structure of the two segments is not reflected.

Table II-3. Software science values for code segments A and B

Segment (A)		Segment (B)	
operators	operands	operators	operands
:=	x	if-then	x
+	a	=	y
f	0	>	0
()	12	:=	12
;	y	f	a
*		()	
$n_1 = 6$	$n_2 = 5$	$n_1 = 6$	$n_2 = 5$
$N_1 = 7$	$N_2 = 6$	$N_1 = 7$	$N_2 = 6$

2. Data dependency measures

The data flow/dependency measures reviewed in this section fall into two categories: intramodule and intermodule. As the names imply, intramodular data flow concerns data usage within a program module, and intermodular data flow concerns the movement of data across module boundaries. Representative measures in each category are discussed.

One of the earliest and simplest intramodular data dependency measures was *span*, defined by Elshoff [34]. A span is the number of statements falling between two references or definitions of the same identifier. Elshoff calculated the span along with other measures in an empirical study of several hundred PL/1 programs. He reasoned that span reflects the locality of variable use and, therefore, provides a measure of the degree of organization of a program. He inferred nothing from his results, however.

Feuer and Fowlkes [8] defined the *mean span* measure as the number of statements between the first reference or definition of a variable and its last reference divided by the total number of references to that variable. A variable referenced in just one statement was defined to have a span of zero. In their experiment described in Section II.B.1, Feuer and Fowlkes compared the mean span measures to their maintenance data, but found no correlation between the two.

Dunsmore and Gannon [22], in their experiment described in Section II.B.2, computed the *average live variables* for each student's program. Computation of this measure begins by determining the number of live variables for each statement in a program. For any statement s, the number of live variables, denoted l(s), is a count of the variables referenced in or before statement s as well as in or after statement s. The researchers conjectured that the effort required to write a program increases exponentially with the number of live variables in the program. So, the average live variables for a program containing n statments is computed as

$$\left(\sum_{s=1}^{n} e^{l(s)} \right)^{\frac{1}{e}}.$$

Dunsmore and Gannon found that the better programs had fewer average live variables than the worse ones, but their results, like Feuer and Fowlkes', were not statistically significant.

Oviedo [35] developed a measure to reflect both control flow and data flow in a module. The measure was defined as

$$C = ad_f + bd_d$$

where a and b are weighting factors to be determined empirically, d_f is the control flow measure, the number of arcs in a program's flowgraph, and d_d is the data flow measure. To compute this last measure, Oviedo first determines all the definition-use pairs in a program. A definition-use pair occurs for a variable x when x is defined in one statement and the result of that definition reaches a reference to x in another statement. The two statements form a definition-use pair. After determining all such pairs, Oviedo eliminates the ones that occur solely within a basic block, and counts the remaining pairs to determine d_d. He tested his measure on a set of program pairs, where one program in each pair was subjectively determined to be "better" than the other member of the pair. His measure reflected the subjective assessment often enough for Oviedo to call the results promising. However, no other empirical studies

of this measure have been reported.

In a quite different approach, Bieman [29] modeled data dependencies using a graph, his *data dependency graph*, or DDG. A data dependency exists in a program when a definition of one variable may depend on values of other variables or on a previous value of itself. For example, in the statement x := f(y,z) the new value of x "depends on" the values of y and z. In the DDG, each node represents a definition, and the arcs between the nodes represent dependencies. The graph for the statement above would contain nodes for all definitions of y and z that could reach the statement, with arcs from these nodes to the node representing the definition of x in the statement. As a measure of a program's data dependency complexity, Bieman computes the number of rooted spanning trees in the program's DDG. In an evaluation using a set of nine program pairs, like Oviedo's, the measure agreed with the subjective ranking for six of the pairs. While this may show that the measure should be evaluated further, the measure has little intuitive appeal; the set of rooted spanning trees does not appear to correspond to any properties found in programs.

While many intramodular data flow measures have been proposed, not nearly as many intermodular measures exist. Only two are reviewed here.

The first was defined by Chapin [36]. His Q measure is computed by counting occurrences of different types of data passed between modules. He defines four types and gives each type a weight factor. They are:

(1) type P data: data used in the computation of a module's output (weight = 1),
(2) type M data: data that are changed or created in the module (weight = 2),
(3) type C data: data used in control flow statements in the module (weight = 3), and
(4) type T data: data items that are neither used nor modified by the module (weight = 0.5).

If a data item satisfies the definition of more than one type, its corresponding weight is the sum of the individual weights for the data types it satisfies.

After determining the types of all the data items transmitted to and from a module, the weights associated with all the data items are summed to produce an intermediate complexity value W'. This value is multiplied by a "repetition factor" R, with the square root of the resulting product taken to obtain Q. The repetition factor reflects iterative invocation of modules, that is, invocation from within loops. It is computed as follows: for every type C data item that is used in an exit test in modules called from within a loop, add two to the iteration-exit factor E (initially set to zero). If that data item is created or modified in some module other than the one containing the loop, then add 1 to E. R is then computed as $R = (E/3)^2 + 1$. Chapin provides no intuition for this formula.

In an example containing only one program, Chapin shows that his Q measure reflects the data flow between modules in the program. He asserts it is better than measures such as McCabe's and Myers' measures (reviewed in Section B) because they do not reflect intermodular sources of complexity. However, no other empirical support for the measure is provided.

The second intermodular data flow measure was proposed by Henry and Kafura [37]. Their *information flow metric* is based on counts of the data items that flow between modules. They defined two types of information flow, global and local. A global flow occurs from a module P to a module Q if P deposits data in a global data structure that is referenced by Q. A local flow occurs when (1) a module P calls a module Q, (2) a module P calls a module Q and Q returns a value to P, and (3) a module R calls modules P and Q passing an output from P to Q. The *fan-in* and *fan-out* of a module are based on these flow types. The fan-in of a module is the number of local flows the module contains plus the number of global data structures from which it retrieves data. The fan-out of a module is the number of local flows the module contains plus the number of global data structures it updates. The information flow metric is computed as $C = \text{length} * (\text{fan-in} * \text{fan-out})^2$. The (fan-

in * fan-out) term represents the total possible combinations of information flow. The product is squared because the authors felt that the complexity due to fan-in and fan-out is more than linear. The length term, the number of statements in the module, reflects the intramodular complexity.

The authors assert that when their measure is applied to procedures in an operating system, it identifies those procedures that (1) may perform more than a single function, (2) may be difficult to modify because of the high degree of interaction with other procedures, and (3) may have been inadequately refined during development. In an experiment in which they compared the complexity values of operating system procedures to the number of changes the procedures had undergone, they found that the $(\text{fan-in} * \text{fan-out})^2$ term to be a good predictor of the probability of changes to a procedure. No further empirical evaluation has been reported.

The review of these measures reveals the importance of considering all the possible properties of software in order to capture its complexity. Focusing solely on one aspect, such as control flow, will not explain every contribution. Although this dissertation considers only control flow measures, the measures presented in this section provide direction for additional research into solving the complexity measurement problem.

D. Deficiencies Common to the Proposed Measures

The measures reviewed in Section B, in addition to their individual shortcomings, possess three major deficiencies. The first is that each measure reflects only one aspect of complexity. As a result, some control flow properties will be ignored; some complexity will be unreported. For example, Gilb's absolute logical complexity [3] produces only the number of predicates in a program. It reveals nothing about nesting, structuredness, or looping. Even the fairly complex scope

number [6] reflects only nesting, and nothing about structuredness or number of paths. This is illustrated by the two flowgraphs in Figure II-5. The unstructured flowgraph (a) has a smaller scope number than the structured flowgraph (b).

The second deficiency is that each measure provides only general information about the property being measured. None provide specific causes or specific locations of complexity in a flowgraph. Consider Gilb's measure again. His simple count of predicates reveals neither the types of the predicates, whether iterative or alternative, nor their relative placement in the program. Similarly, the scope number gives only a general degree of nesting in a flowgraph. It does not reveal whether the flowgraph contains many shallowly nested constructs or a few deeply nested ones. As a result, the measure cannot pinpoint specific instances of deep nesting. This type of information is needed for identifying potential problem areas in programs.

The third, and most significant, deficiency is the lack of evaluation of the proposed measures. Except for McCabe's cyclomatic complexity [2], the reviewed measures have received little empirical testing. Further, many of the attempts at empirical evaluation have been inadequate. For example, Feuer and Fowlkes [8] tested the utility of two path-based measures, but instead of actually counting paths in the modules they examined, they estimated the counts. Additionally, they tried to correlate the measure values with a set of programmer time data that they admitted was incomplete and not wholly accurate. In Schneidewind and Hoffman's experiment [16], Hoffman was both experimenter and subject, an arrangement that may have unintentionally biased the results. Valid empirical evaluations must be free of these types of problems.

Analytical evidence in support of the measures is weaker than empirical evidence. The definitions of control flow properties and their measures contain little rigor. Both Feuer and Fowlkes [8] and Schneidewind and Hoffman only cursorily defined their notions of paths. The structuredness measures do not, in general, reflect

the true degree of structuredness of a flowgraph. Harrison and Magel [6] provided only intuitive definitions of greatest lower bounds and scope number, leaving formalization to the reader. A quick analysis of Chen's nesting measure reveals that Chen [4] made it so general that it no longer reflects nesting at all. When measures are as poorly defined and analyzed as these, poor empirical results should be no surprise.

Many of the flaws in the empirical work also arise because the actual causes of psychological complexity in programs have not been accurately identified. Many researchers appear to be relying too much on intuition and not making an effort to obtain hard psychological evidence. Intuition is also largely to blame for the lack of analytical evidence. The attitude that "everyone knows what a loop, nesting, and structuredness are" significantly hinders the production of well-defined measures.

However, intuition cannot be ruled out altogether. A measure of node nesting appears, *intuitively*, to be more useful than a count of rooted spanning trees in a flowgraph. But, once an intuitively satisfying property has been identified, mathematical rigor and analytical evaluation techniques must be used to prove the worth of the property. This idea motivates the methods used in the research reported in the following chapters.

III. NESTING AND STRUCTUREDNESS

Statement nesting has long been considered a prime contributor to program complexity. Weissman, in his paper on the causes of complexity in programs [7], said that excessive nesting can make a program unintelligible. Weinberg felt so strongly against nesting that he defined constructs that would allow the expression of deeply nested *if* statements as singly-nested constructs [9].

Nesting increases the difficulty of comprehending a program by requiring the programmer to understand the conjunction of all the individual conditions which predicate the execution of a given nested statement. When performing maintenance tasks, the programmer must be able to identify all the predicators of a given statement to determine how control reaches that statement. In structured programs coded in high-level languages, finding these predicators is not difficult. But in unstructured programs and programs coded in low-level languages, the predicators are not as easily identified. A tool that enumerates all the predicators of a given statement would be a valuable aid to understanding the control structure of these programs. A count of the predicators of each statement or basic block would pinpoint instances of deep nesting, as well as provide quality information about the code. Thus, the first element in the set of properties used to characterize the control flow surrounding a node is the set of predicators of that node. The corresponding measure is the count of those predicators.

Although this first property identifies all the predicators of each node in a flowgraph, it reveals nothing about the nature of those predicates. The context of a node nested in a series of *if* statements is significantly different than the context of the same node nested in a series of *repeat* statements. In the first case, the programmer only has to determine the conditions in the *if*'s that cause control to reach a given node. But in the second, besides needing to identify the conditions, the programmer must identify those predicates whose conditions could be affected by

executing the given node. The exact nature of the effect is not a control flow issue, but a data dependency issue, as discussed in [29, 22, 38]. As further evidence, in program proof techniques, determining the weakest preconditions for alternative constructs is often much easier than deriving loop invariants. Thus, differentiating iterative and alternative predicates is warranted. The second property, the property that characterizes this difference, is the set of nodes that predicate loops in which a given node is nested. Its corresponding measure is the count of such nodes.

Before definitions can be derived for these first two properties, the notion of nesting must be formalized. Section A presents definitions of nesting and of the first two properties, which depend heavily on the nesting definition. In Section B, the measures are analytically evaluated to demonstrate that they accurately reflect their corresponding properties.

Receiving as much attention as the nesting issue is the structuredness issue. Since Dijkstra pointed out that the indiscriminate use of *goto* statements increases the complexity of programs [39], most computer scientists have agreed that regularity of program structure is a highly desirable property. They believe that this regularity is best achieved through structured programming, building programs by sequencing and nesting the elementary structured constructs defined in Chapter II. The degree of structuredness of the control flow surrounding a given node provides an indication of the regularity of that code. Thus, the third element of the set of characterizing properties is the structuredness of the code containing a node. Both this property and its corresponding measure are defined in Section C.

A. Determining Predicators

The formal definition of nesting is derived from the work by Harrison and Magel on scope number [6]. As explained in the review in Chapter II, they took the view that the complexity of a predicate node is directly tied to the complexity of the nodes that it predicates. However, when considered from the perspective of the

programmer, this seems inverted. Knowing what conditions must be true for control to reach a given statement seems more useful than knowing the statements that are executed depending on the truth value of a given predicate. In a program composed solely of elementary structured constructs, nesting is well-defined and identifying the predicators of a given node is fairly simple. In an arbitrary flowgraph, however, nesting is not as obvious and identifying a node's predicators is more difficult. The following section defines a method of determining nesting. Sections 2 and 3 use that definition to develop the first two desired properties.

1. A definition of nesting

Nesting can be defined by making more rigorous Harrison and Magel's notion of the range of a predicate. Intuitively, the range of predicate node p is the set of nodes whose execution may be determined by the truth value of p. Once the ranges of the predicates in a flowgraph are identified, the nesting of an arbitrary node x can be determined by identifying the ranges in which x falls. The following definitions rigorously capture the notion of range. The first defines a necessary restriction on paths in a flowgraph.

> *Definition 3.1:* In a flowgraph G, a *first occurrence path* from node x to node y, *FOP(x,y)*, is a path from x to y such that node y occurs exactly once on the path.

The intent here is to consider only the paths from x to y that do not contain cycles involving y. This restriction is integral for the development of the definition of range.

The nodes lying on specific first occurrence paths constitute the range of a predicate. The next definition provides a means to reference these nodes as a set.

> *Definition 3.2:* The set of nodes that fall on any first occurrence path from node n to node m, the members of the path, MP, is:

$$MP(n,m) = \{v \mid \exists P\,[P = FOP(n,m) \wedge v \in P]\}.$$

The next definition simply provides some needed terminology.

Definition 3.3: In a flowgraph, the set of immediate successors of a node n is denoted by IS(n), where

$$IS(n) = \{m \mid (n,m) \in E\}.$$

When a predicate node splits one path into two or more, there is at least one node in the flowgraph where all these paths merge back together. This node is used to determine the extent of a range of a predicate. The node is contained in a set of nodes common to all paths from the immediate successors of the predicate to the exit node. This set is called the *lower bounds* of the predicate.

Definition 3.4: In a flowgraph G = (N, E, s, t), the set of lower bounds of a node n is given by

$$LB(n) = \{v \mid \forall r \forall P\,[r \in IS(n) \wedge P = FOP(r,t) \Rightarrow v \in P]\}.$$

The lower bounds of n are exactly the inverse proper dominators of n. That is, if a flowgraph G′ is created from flowgraph G by reversing every arc in G, then the lower bounds of n in G are exactly the proper dominators of n in G′. Since every node in LB(n) lies on every path from n to t in G, then the same nodes must lie on every path from t to n in G′. The unique immediate dominator of n in G′ is called the greatest lower bound of n in G.

Definition 3.5: The unique lower bound that precedes all other lower bounds is called the *greatest lower bound*, *GLB*. Formally,

$$GLB(p) = q \mid q \in LB(p) \wedge \forall r\,[r \in (LB(p) - \{q\}) \Rightarrow r \notin MP(p,q)].$$

When a predicate node p is executed, then GLB(p) is also executed. The execution of GLB(p) does not depend on the truth value of p because GLB(p) lies on every path from p to t. Since GLB(p) is the first node encountered on paths from p to t for which this holds, the execution of every node lying on paths between IS(p)

and GLB(p) must depend on the truth value of p. These nodes, called the range of p, are considered to be nested within the construct predicated by node p.

Definition 3.6: Range(p) is the set of nodes predicated by node p, those nodes that fall on any path from the immediate successors of p to the greatest lower bound of p. Formally,

$$\text{Range}(p) = \{n \mid \exists q\,[q \in \text{IS}(p) \wedge n \in \text{MP}(q,\text{GLB}(p))]\} - \{\text{GLB}(p)\}.$$

Range is defined not only for predicate nodes, but also for nonpredicate nodes. However, the range of a nonpredicate node x is empty because IS(x) = {GLB(x)}.

In the definition of structuredness, it is often convenient to refer to a predicate node and its range as a single set, the range′ of the predicate.

Definition 3.7: $\text{Range}'(p) = \text{Range}(p) \cup \{p\}$.

These definitions provide a basis for defining the nesting and loop properties described in Sections 2 and 3.

Integral to the development of the definition of structuredness is the nesting of ranges. The following lemma and theorem reveal important properties about nested ranges. Intuitively, the lemma shows that if a predicate p is nested in Range(q), then GLB(p) must occur either before or at the same time as GLB(q) on every path from p to t.

Lemma 3.1: If $p \in \text{Range}(q)$, then $\forall P[\, P = \text{FOP}(p,\text{GLB}(q)) \Rightarrow \text{GLB}(p) \in P]$.

Proof: Every path from p to t must contain GLB(p). Since $p \in \text{Range}(q)$ and every path from q to t contains GLB(q), then every path from p to t must contain GLB(q). This implies, directly from the definition of lower bounds, that $\text{GLB}(q) \in \text{LB}(p)$. Since GLB(p) precedes every other lower bound of p, GLB(p) must be on every path from p to GLB(q). □

This lemma is used to establish a more general result: if a predicate p falls within Range(q), then every node of Range(p) must also fall within Range(q).

Theorem 3.1: If $p \in \text{Range}(q)$, then $\forall x[\, x \in \text{Range}(p) \Rightarrow x \in \text{Range}(q)]$.

Proof: If p is a nonpredicate node, then Range(p) is empty, and the theorem is vacuously true. Suppose p is a predicate node. Let $x \in \text{Range}(p)$. It must be shown that x lies on some FOP(q,GLB(q)). There exists a path from q to p since $p \in \text{Range}(q)$. There exists a path from p to x since $x \in \text{Range}(p)$. In the proof of Lemma 3.1, every path from p to t is shown to contain GLB(q), including those paths that contain x. By the definition of a flowgraph there must be at least one such path. Therefore, the required FOP(q,GLB(q)) containing x is constructed. □

As a result of this theorem, $p \in \text{Range}(q)$ implies $\text{Range}(p) \subseteq \text{Range}(q)$. The containment is not necessarily proper. For instance, suppose nodes p and q predicate the same loop. Then $p \in \text{Range}(q)$ and $q \in \text{Range}(p)$. Therefore, $\text{Range}(p) \subseteq \text{Range}(q)$ and $\text{Range}(q) \subseteq \text{Range}(p)$, which implies $\text{Range}(p) = \text{Range}(q)$.

2. A property of predicators

Using the definition of range, the predicators of a given node n can easily be identified.

Definition 3.8: The set of nodes that predicate node n is:

$$\text{Pred}(n) = \{p \mid n \in \text{Range}(p)\}.$$

By using Pred(n), the degree of nesting of node n can be quantified by computing its *predicated execution number*, *PEN*.

Definition 3.9: For any node n in flowgraph G,

$$\text{PEN}(n) = |\text{Pred}(n)|.$$

Both Harrison and Magel's scope number, SN, and the predicated execution number can be defined for an entire flowgraph G using the range definitions. (Since the concern here is strictly control flow, in computing the scope number, each node's raw complexity value is assumed to be 1, as in [6].) For any flowgraph G = (N, E, s,

t), SN(G) and PEN(G) are defined in the next two equations.

$$SN(G) = \sum_{n \in N} \left| 1 + |Range(n)| \right|$$

$$PEN(G) = \sum_{n \in N} |Pred(n)|$$

Theorem 3.2: For a given flowgraph G = (N, E, s, t), SN(G) = PEN(G) + |N|.

Proof: It suffices to show that each node in G contributes the same value to PEN(G) as it does to SN(G). Let $v \in N$. Let l(v) be the number of predicate ranges in which v falls, that is, l(v) = |Pred(v)|. Let r(v) denote |Range(v)|.

Consider the contribution of node v to SN(G). It directly contributes 1, its raw complexity value, plus r(v), 1 for each node falling in Range(v). It indirectly contributes 1 to the scope number of every predicate in whose range it falls, for a total indirect contribution of l(v). Thus, the total contribution to SN(G) of node v is 1 + r(v) + l(v).

Now consider node v's contribution to PEN(G). It directly contributes l(v), 1 for each predicate in whose range it falls. It indirectly contributes 1 to the predicated execution number of each of the r(v) nodes in Range(v). Thus, the total contribution of node v to PEN(G) is l(v) + r(v). For the PEN, the direct contribution of every node is the sum of indirect contributions of some other nodes. It follows that the sum of the direct contributions is equal to the sum of the indirect contributions. Thus,

$$\sum_{n \in N} r(n) = \sum_{n \in N} l(n)$$

$$\sum_{n \in N} (1 + r(n)) = \sum_{n \in N} (1 + l(n))$$

$$\sum_{n \in N} \left| 1 + |Range(n)| \right| = \sum_{n \in N} 1 + \sum_{n \in N} |Pred(n)|$$

$$SN(G) = PEN(G) + |N| \quad \square$$

This result demonstrates that the scope number and the predicated execution number, when applied to an entire flowgraph, are quantitatively equivalent and just

the perspective from which they are viewed differs.

3. A property of containing loops

To identify the predicates of loops that contain a given node, the following property relating cycles and ranges first must be established.

Theorem 3.3: Let G = (N, E, s, t) be a flowgraph. Let $p \in N$. Then $p \in Range(p)$ iff p predicates a cycle in G.

Proof: $\Rightarrow$: Suppose $p \in Range(p)$. Then, there exists at least one path in Range(p) from $x \in IS(p)$ to GLB(p) that contains p. Therefore, there exists at least one cycle from p to p. There must also exist a path from p to GLB(p) that does not contain any nodes other than p on at least one cycle from p to p. By way of contradiction, assume that this is not true. Then, there must be one node, y, common to all the cycles, through which all paths from p to GLB(p) pass. Then, y must be GLB(p). But, GLB(p) cannot lie on the cycles because p does not predicate GLB(p). Hence, there must be a path from p to GLB(p) containing no nodes, but p, on some cycle. Node p predicates that cycle.
$\Leftarrow$: Let p predicate a cycle C. Then, there exists a path from an immediate successor of p to t that does not contain any nodes in the cycle. GLB(p) cannot lie on C, because p does not predicate GLB(p). But, GLB(p) lies on every path from p to t. Therefore, there exists a path from an immediate successor of p through p to GLB(p). This implies $p \in Range(p)$. □

By the theorem, a node n falls within the range of a loop predicate p if and only if both n and p are in Range(p). However, not all nodes that fall within the range of a loop predicate are actually part of a loop. Consider the flowgraph in Figure III-1. Node z falls within Range(p2), but there is no path from z to p2 fully contained in Range(p2). Therefore, node z cannot be considered to lie on the loop predicated by node p2. (It does, however, lie on the loop predicated by node p1.)

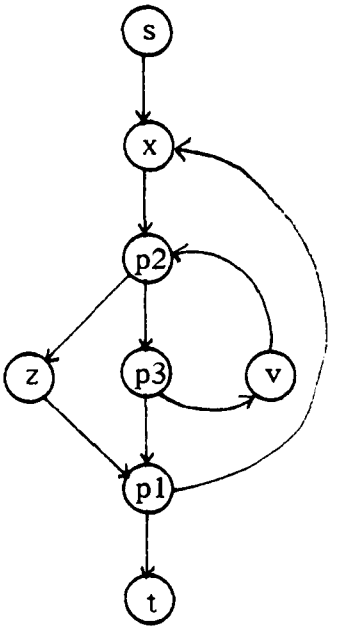


Figure III-1. Node in loop range but not in loop

Intuitively then, a node x lies in a cycle delimited by node p if x falls in Range(p) and there is a path from x to p that contains only nodes in Range(p).

Definition 3.10: The set of loop predicates delimiting cycles that contain node x, LPred(x), is given by:

$$LPred(x) = \{p \mid x \in Range(p) \wedge \exists P\,[P=FOP(x,p) \wedge P \subseteq Range(p)]\}\ .$$

The measure reflecting this property is the cardinality of LPred.

Definition 3.11: The number of loop predicates that delimit cycles containing node n, NLP(n), is

$$NLP(n) = |LPred(n)|\ .$$

For any node $v \in N$, $PEN(v) \geq NLP(v)$. This follows because the number of loop predicates deciding a node v can never exceed the total number of predicates deciding v.

B. Evaluation of PEN and NLP

This section provides evidence that the PEN and NLP measures accurately quantify the nesting properties Pred and LPred. Since the measures are simply the cardinalities of Pred and LPred, this evaluation will also validate the properties. To evaluate the measures, a nesting measure evaluation tool must be defined. The tool should satisfy the following criteria:

(1) It should be general enough to evaluate any proposed nesting measure.

(2) It should test for the difference in nesting of alternative and iterative constructs.

(3) Nesting should be the only control flow variable, other than alternation and iteration.

The tool proposed to meet these criteria is a set of six flowgraphs. Each flowgraph contains n predicates, p_1 through p_n, and n basic blocks, b_1 through b_n. Block b_i is immediately nested within Range (p_i) in every flowgraph. Any complexity inherent in predicate p_i and basic block b_i is assumed to be identical in every flowgraph.

Iteration and alternation are represented by *while* and *if-then* constructs, respectively. Each flowgraph contains just one type of elementary construct. These particular constructs were chosen because they are highly symmetric. In each, the predicate must be executed first, and each contains exactly one immediately nested basic block. The six flowgraphs are constructed by sequencing and nesting these elementary constructs.

The structure of the flowgraphs represents the extremes of nesting—total sequencing and maximal nesting—plus a nesting structure between the two. The minimally nested flowgraphs, the sequential alternative (SA) and sequential iterative (SI) flowgraphs, are constructed by placing the n elementary constructs in sequence; no predicate falls within the range of another. Flowgraphs for the other nesting

extreme, represented by the maximally nested alternative (MNA) and maximally nested iterative (MNI) flowgraphs, are formed by nesting predicate p_i in Range(p_{i-1}) for $1 < i \leqslant n$. Representing the "in-between" nesting are the intermediate nested alternative (INA) and intermediate nested iterative (INI) flowgraphs. These are constructed by nesting predicate p_i in Range($p_{i/2}$), Range($p_{i/4}$), ..., Range(p_1), for $1 < i \leqslant n$. Because of the way these last two flowgraphs are constructed, node b_i is nested within $\log_2(i)$ predicates. Since fractional predicates do not make sense, $\log_2(i)$ is taken to mean floor($\log_2(i)$). The flowgraphs of these six constructs are shown in the Appendix.

Table III-1 summarizes the PEN and NLP values for the basic blocks and predicates in each flowgraph. The table shows the following relationships. In the SA construct, no predicate is nested and each basic block is nested only within its corresponding predicate's range. The NLP values reflect the absence of loops. In the SI construct, each predicate and its corresponding basic block are nested in the range of the predicate. The PEN and NLP measures reflect this.

In the INA construct, predicates p_8 and p_9 are nested in Range (p_1), Range(p_2), and Range(p_4). Since $\log_2(8) = \log_2(9)$ the PEN values reflect the identical nesting levels. The same holds for basic blocks b_8 and b_9. However, p_7 is contained only in Range(p_1) and Range(p_3); it is at a shallower nesting level than p_8. Since

Table III-1. PEN and NLP values for the nodes in nesting evaluation tool flowgraphs

Flowgraph	PEN(p_i)	PEN(b_i)	NLP(p_i)	NLP(b_i)
SA	0	1	0	0
SI	1	1	1	1
INA	$\log_2(i)$	$\log_2(i)+1$	0	0
INI	$\log_2(i)+1$	$\log_2(i)+1$	$\log_2(i)+1$	$\log_2(i)+1$
MNA	i-1	i	0	0
MNI	i	i	i	i

$\log_2(7) < \log_2(8)$, the PEN values reflect this difference. The same holds for both the PEN and NLP values in the INI construct.

In the MNA construct, for $1 \leqslant i < j \leqslant n$, $PEN(p_i) < PEN(p_j)$ and $PEN(b_i) < PEN(b_j)$, accurately reflecting the nesting level differences of the individual ranges. The same hold for the MNI construct, where both the PEN and NLP values reflect the additional predicate nesting caused by iteration.

These relationships show that the PEN and NLP measures accurately reflect the differing nesting levels and their causes in these flowgraphs.

In addition to their utility as a nesting evaluation tool, the six flowgraphs possess some interesting properties. They can be ranked according to the partial ordering "contains less nesting than" (as given by the sum of the PEN values of all the nodes). This ordering is represented by the lattice in Figure III-2. The ranking shown in the lattice follows because of the difference in nesting levels or in the elementary constructs comprising the flowgraphs, or both. The SA construct contains the least nesting, and the MNI construct contains the most; thus, they are the bottom and top elements, respectively. No general comparison can be made between INI and MNA flowgraphs because how the loop nesting in INI compares to the maximally nested *if-then* constructs depends on i. The same applies to SI and INA.

The PEN and NLP values of corresponding nodes in each flowgraph reflect the same ranking as in the lattice (see Table III-1). Other comparisons also can be made by measuring at the node level. First, because of the way in which the flowgraphs are built, the following can easily be seen:

$$PEN_{SA}(b_i) = PEN_{SI}(b_i)$$

$$PEN_{INA}(b_i) = PEN_{INI}(b_i)$$

$$PEN_{MNA}(b_i) = PEN_{MNI}(b_i).$$

These equalities reflect that each basic block is nested at the same level in corresponding flowgraphs.

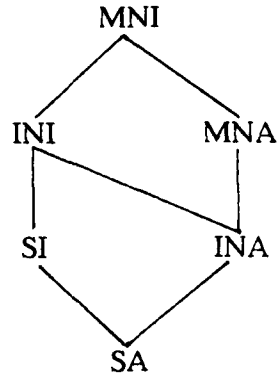


Figure III-2. Partial ordering by nesting

Secondly, the inequalities

$$PEN_{SA}(p_i) < PEN_{SI}(p_i)$$

$$PEN_{INA}(p_i) < PEN_{INI}(p_i)$$

$$PEN_{MNA}(p_i) < PEN_{MNI}(p_i)$$

reflect that each predicate is nested within its own range in the iterative constructs, but not in the corresponding alternative constructs.

Third, the inequalities

$$PEN_{SA}(b_i) \leqslant PEN_{MNA}(b_i) \leqslant PEN_{MNA}(b_i)$$

$$PEN_{SI}(b_i) \leqslant PEN_{MNI}(b_i) \leqslant PEN_{MNI}(b_i)$$

reflect the difference in the nesting levels within the same elementary construct class. These follow directly from the construction of the flowgraphs. In both the alternative and iterative flowgraphs, the first (leftmost) inequality becomes strictly less than for $i > 1$; the second becomes strictly less than for $i > 2$.

As a final observation, $NLP(b_i) = NLP(p_i) = PEN(p_i)$ in all of the iterative constructs. And as expected, $NLP(p_i) = NLP(b_i) = 0$ in each of the alternative constructs.

C. Structuredness

This section presents the definition of the third property that helps characterize control flow, the degree of structuredness of a given node's containing constructs. As the review of the structuredness measures in Section II.B.4 revealed, this property cannot be based solely on identifying the proportion of a flowgraph that contains elementary structured constructs. Structured constructs cannot be identified easily when they are mixed with unstructured ones. However, useful information can be obtained from a brief analysis of the structured constructs and from an analysis of the observations of McCabe and Williams.

The first obvious property of the elementary structured constructs is that they possess single, well-defined entry points. Any construct with two or more entry points could be considered unstructured. It should be noted that these constructs are said to possess single, well-defined exit points, also. However, because of the method used to generate flowgraphs, an *if* construct actually has two exit points. Thus, the single-exit property exists solely in the linearization, or coding, of the constructs, where a single terminal point of structured statements can be easily identified.

The second property of the elementary structured constructs is that they are singly-predicated structures. This implies that any construct predicated by two or more decision nodes is unstructured. This is the view used in the development of the structuredness property, although advocates of the careful use of multiple-exit loops [11, 10] contend that they are just as structured as single-predicate loops. Extending the structuredness definition to include single-level multiple exit loops is not difficult, but attempting to include all possible forms of multilevel-exit loops compounds immensely the problem of determining structuredness. For this reason, only basic structured constructs are recognized as being structured. Extensions are left for future research.

The most important observation made, by McCabe [2] and Williams [28], independently, was that the interaction of at least two predicates is required to produce an unstructured construct. This implies that if the predicates in a flowgraph are considered pairwise, a check of how the predicates in each pair interact can reveal their relative structuredness. The degree of interaction can be determined by examining how their ranges overlap. Thus, structuredness can be based on predicate range information.

McCabe also observed that an unstructured construct contains both a multiple-entry and a multiple-exit structure. One never appears without the other. Thus, although a multiple-exit construct (in the unstructured sense) is very difficult to detect in a flowgraph, identification of a multiple-entry construct is reasonably simple and will reveal its corresponding multiple-exit part. Thus, only one of the properties needs to be tested for to detect the presence of both.

Applying these observations provides the first condition for two predicates to be structured with respect to each other:

> If the ranges of two predicates do not overlap, if the intersection of the ranges is empty, then the predicates are structured with respect to each other.

This follows immediately from the first observation of McCabe and Williams. If two predicates' ranges do not overlap, their control flow cannot interact; thus, they cannot form an unstructured construct.

The single-entry property of elementary structured constructs leads to the next condition for structuredness. Assume node p predicates a single-entry construct. Then all paths that enter Range'(p) must do so at its unique entry node. This observation provides the "all paths" principle:

> Two predicates are structured with respect to each other if all the paths from the first into the range' of the second enter that range' at exactly one node.

This principle seems to provide a fairly simple method of testing for structuredness. It infers that if there are paths from some predicate p that enter the range' of a predicate q at two distinct nodes, then p and q are unstructured with respect to each other. However, this interpretation is too restrictive. Consider the flowgraph in Figure III-3. Paths originating at predicate p1 enter Range'(p4) at nodes c and p4, violating the all-paths principle. But the actual structure problem is caused only by the interaction of predicates p3 and p4. Thus, a definition of structuredness should define predicates p3 and p4 to be unstructured with respect to each other, and define all of the rest of the predicate pairs as structured. The intent is to minimize the number of pairs that are defined to be unstructured; the unstructuredness of one pair should not propagate to other pairs in the context. (This is exactly the deficiency

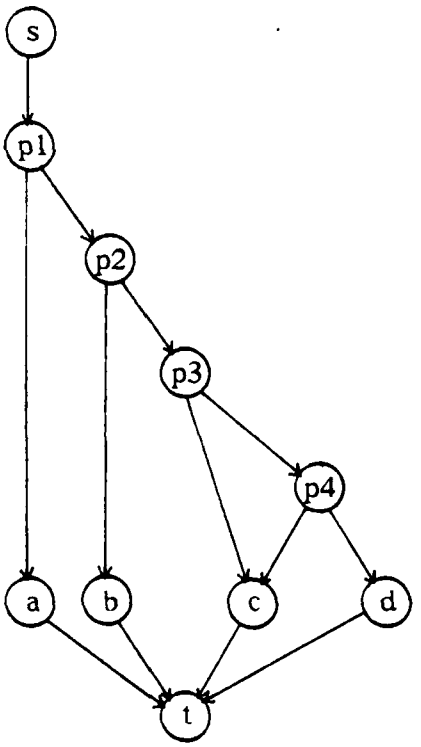


Figure III-3. A violation of the all-paths principle

of the flowgraph reduction measure reviewed in Section II.B.4.) Conversely, the definition of structuredness should account for all the "well-structured" components, viewing the components as pairs. This objective can provide the programmer an optimal structured view of an arbitrary program.

Since unstructuredness can occur only when two predicate ranges overlap in some way, the only acceptable form of overlapping must be "structured nesting". Two requirements must be satisfied for two predicates to be nested in a structured fashion. First, the Range' of one predicate must be wholly contained within the Range' of the second, and that containment must be proper. The following two cases illustrate the need for this first requirement. In the first case, shown in Figure III-4, the ranges of predicates p2 and p3 share node b, but neither's range' is fully nested in the other's. The result is that both Range'(p2) and Range'(p3) are multiple-entry, and are, therefore, unstructured with respect to each other. The second case involves detecting constructs with multiple predicators, specifically the multiple-exit loop. The loop forms a single construct, and since each of the loop's predicates lie in the

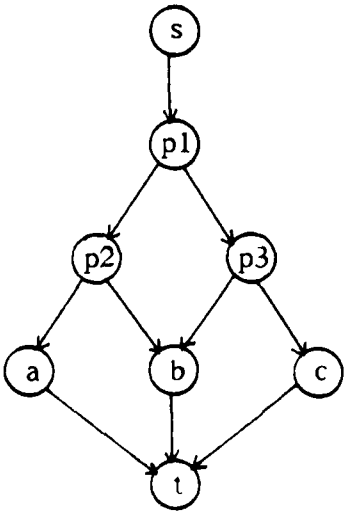


Figure III-4. Flowgraph containing overlapping ranges

construct, the range of each predicate contains each of the other loop predicates. Thus, all of the ranges are equal; there is no proper containment of the ranges.

The second requirement for predicates to be nested in a structured fashion is derived from the all-paths principle: all paths from the containing predicate must enter the range' of the contained predicate at only one node, the *entry node*. The entry node of a Range'(p) is denoted as Entry(p).

Thus, if Range'(q) is single-entry and is totally contained in Range'(p), it must be properly nested in Range'(p), because every path from p into Range'(q) can enter only at Entry(q). The notion of structured nesting for a multiple-entry construct is not defined as easily because the choice of entry node for the nested construct is not as obvious. This is discussed later.

Additional analysis of the elementary structured constructs reveals that the entry node for any single-entry construct predicated by node p is the node that is contained in and dominates every node in Range'(p). Dominance occurs because if a construct is single entry, every path from s to any node in the construct must contain the entry node. However, having the dominator of Range'(p) fall within Range'(p) does not guarantee that Range'(p) is single-entry. In the flowgraph in Figure III-5, Range'(p1), nodes p1, x, and y, is nested within Range'(p2). Node x dominates Range'(p1) and is, therefore, the entry node. But there is a path from p2 into Range'(p1) that does not include node x, making Range'(p1) a multiple-entry construct.

Now reconsider the construct in Figure III-3. The paths from node p2 into Range'(p4) enter at two distinct nodes, violating the all-paths principle. But as noted earlier, the unstructuredness is caused only by predicates p3 and p4. Predicates p2 and p4 should be considered to be structured with respect to each other. Observe that all the paths from p2 into Range'(p4) contain node p3. This implies that the split in the path causing the multiple entries into Range'(p4) must occur within Range'(p3).

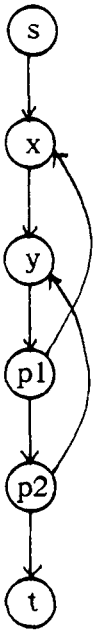


Figure III-5. Multiple entry construct containing its dominator

No control structure immediately within Range'(p2) causes any additional paths. This observation can be incorporated into the all-paths requirement, leading to the following extension:

> Assume Range'(p2) is properly nested in Range'(p1). If there is a predicate p3 such that i) Range'(p3) is properly nested in Range'(p1), ii) Range'(p2) is properly nested in Range'(p3), and iii) every path from p1 into Range'(p2) contains Entry(p3), then p1 is structured with respect to p2.

In Figure III-3, predicate p3 corresponds to the predicate p3 required in the extension. By this extended all-paths principle, the unstructuredness is localized to predicates p3 and p4 and is not propagated to predicates p1 and p2.

The extended requirement does not specify that Range'(p3) be single-entry, only that Entry(p3) fall on all paths leading from p1 to Range'(p2). In fact, the entry node referred to in the all-paths principle need not be the only entry point into Range'(p3). This implies that not every predicate that is the source of paths into

Range'(p) should be considered unstructured with respect to p. Specifying a node as Entry(p) allows some of those predicates to be considered structured. This leads to the problem of identifying one entry node for a multiple-entry construct as "the" entry node for that construct. Applying the objective of minimizing the number of unstructured pairs and providing an optimal view of the constructs, Entry(p) should be chosen so it maximizes the number of predicates defined to be structured with respect to p. Since the basis for structuredness is having all paths from a containing predicate enter Range'(p) at a single node, Entry(p) can be the node through which the most predicates have all paths entering Range'(p). Thus, if Range'(p) has candidate entry nodes e_1, e_2, ..., e_k, then Entry(p) is the e_i that maximizes the number of nodes in the set:

$\{ q \mid p \in \text{Range}'(q)$ and every path from q into Range'(p) enters at node $e_i \}$.

Those predicates that have paths entering Range'(p) at other than e_i will then be declared unstructured with respect to p. This definition, based on the all-paths principle, should select the entry node that optimizes the number of structured pairs for all multiple-entry constructs, both alternative and iterative; but it does not.

Consider first multiple-entry alternative constructs. Specifically, consider the construct predicated by p2 in the flowgraph of Figure III-6. Range'(p2) = {p2, a, b}. The candidate entry nodes for Range'(p2) are p2 and b. No predicate has all paths entering Range'(p2) at node p2, but all paths from p3 and p4 enter at node b. So, to optimize the number of structured pairs, node b should be chosen as Entry(p2), making p3 and p4 structured with respect to p2. But Range'(p3) and Range'(p4) overlap Range'(p2), a condition defined to be unstructured. Thus, selecting node b as Entry(p2) contradicts other requirements for structuredness.

That the selection process does not work for alternative constructs is not surprising. It is intuitively unappealing to consider anything but the predicate node of an alternative construct as its entry point. The choice of entry node should ensure

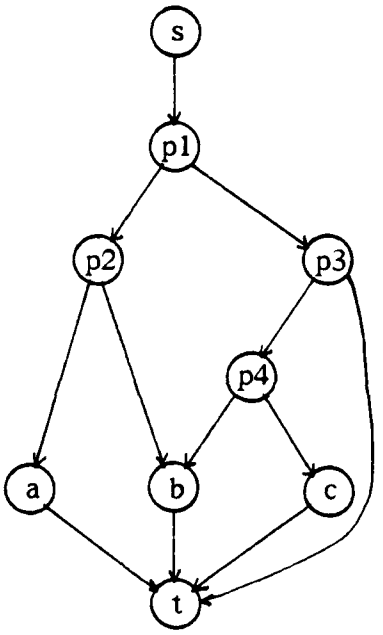


Figure III-6. Multiple-entry alternative construct

that every other node in the range' of a predicate is reachable from its entry node. This holds for alternative constructs only if the predicate itself is chosen, unless there exists another node in the range' that dominates the range'. (For example, in Figure III-7, node p2 forms an alternative construct, but x dominates Range'(p2) and is, therefore, the reasonable choice for Entry(p2).)

Unlike for alternative constructs, the entry node selection process works well for multiple-entry iterative constructs, and does so for three reasons. First, regardless of the choice of entry node, there will always be a path from the entry node to all of the rest of the nodes in the loop. Secondly, loop ranges cannot overlap in the way alternative ranges can. If two loops share a node, then each loop will fall in the other's range'. Thirdly, if an entry node y is chosen because all paths from predicates p1, p2, ..., pk enter the loop at y, then the all-paths principle guarantees that those predicates will be structured with respect to the loop's predicate(s).

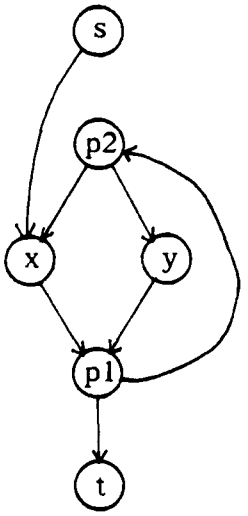


Figure III-7. Alternative construct with nonpredicate dominator

The following definition formalizes the notion of entry point developed above.

Definition 3.12: Entry(p), the entry node of Range'(p), is determined as follows:

If there is a node x such that
  x is in Range'(p) and
  x dominates every node in Range'(p)
 then Entry(p) = x

else if p is an alternative predicate
 then Entry(p) = p

else
 Entry(p) = node e in Range'(p) where e maximizes
 |{q | q ∈ Pred(p) ∧ every path from q into Range'(p) contains e}|.

 (If several e_i produce the same maximum value,
 then arbitrarily select one of them.)

The all-paths and extended all-paths principles can be combined with the definition of entry node to define structuredness.

Definition 3.13: For predicates p and q, let Struct (p, q) denote that p is structured with respect to q, and Unstruct (p, q) denote otherwise. The following procedure determines the relative structuredness of p and q:

```
{1} If Range'(p) ∩ Range'(q) = ∅ then
     Struct(p, q)

   else
{2} if p ∈ Range'(q) and q ∈ Range'(p) or
       p ∉ Range'(q) and q ∉ Range'(p) then
     Unstruct(p, q)

   else { Assume WLOG that q ∈ Range'(p). }
{3} if every path from p into Range'(q) contains Entry(q)
     then Struct(p, q)

   else
{4} if there exists a predicate r such that
       Range'(r) ⊂ Range'(p) and
       every path from p into Range'(q) contains Entry(r)
     then Struct(p, q)

   else
     Unstruct(p,q).
```

The degree of unstructuredness in the control flow surrounding a given node can be quantified as the ratio of the unstructured predicate pairs to the total number of predicate pairs in that node's set of predicators. This is formalized by the following definitions.

Assume the predicate nodes in flowgraph G are ordered in some fashion. Let P be the set of predicate nodes in N, $P = \{p_1, p_2, ..., p_k\}$.

Definition 3.14: The predicate pairs in a set of nodes S is:

$$Pairs(S) = \{(p_i, p_j) \mid p_i, p_j \in (P \cap S) \wedge i < j\}.$$

This definition ensures that each distinct pair of predicates is considered only once.

Definition 3.15: The set of unstructured pairs in the control flow context of node x is:

$$Unstr(x) = \{(p_i, p_j) \mid p_i, p_j \in Pairs(Pred(x)) \wedge Unstruct(p_i, p_j)\}.$$

Definition 3.16: The degree of unstructuredness of the control flow context of node x is:

$$UNST(x) = \frac{|Unstr(x)|}{|Pairs(Pred(x))|}.$$

Chapter V contains algorithms that implement the range, PEN, NLP, and structuredness definitions. As shown there, even a straightforward implementation of the definitions is computationally feasible.

The following set of theorems provide not only additional insight into the definition of structure, but also some direction into implementing software that checks program structure. In the theorems, Struct(p,q) denotes "p is structured with respect to q" and Unstruct(p,q) denotes "p is unstructured with respect to q."

Theorem 3.4: If Range'(p) does not contain a node that dominates Range'(p), then there exists a predicate q such that Unstruct(p,q).

Proof: Since Range'(p) does not contain its immediate dominator, it must be multiple entry. Since there are paths entering Range'(p) at two distinct nodes, there must be a predicate node that causes the original single path from the start node to split into the multiple paths that enter Range'(p). Call this predicate node q. Since there exist paths from q entering Range'(p) at two separate nodes, Unstruct(p,q). □

Theorem 3.5: If $p \in$ Range'(q) and Struct(p,q), then for all predicates s in Range'(p), Struct(s,q).

Proof: Since $p \in$ Range'(q) and Struct(p,q), all paths from q into Range'(p) enter through Entry(p). Thus, every path into Range'(s), where $s \in$ Range'(p), must also pass through Entry(p). Therefore, for each predicate s, p is the predicate p3 required in the structure definition. Therefore, Struct(s,q). □

Theorem 3.6: If Unstruct(p,q) and $p \in$ Range'(q), then for all predicates r such that $p \in$ Range'(r) and $r \in$ Range'(q), Unstruct(r,q).

Proof: There are two cases to consider:

(1) Range'(p) = Range'(q): Then any r satisfying the above conditions will have to be such that Range'(r) = Range'(q), resulting in Unstruct(r,q) by definition.

(2) Range'(p) $\subset$ Range'(q). There must be paths that start at node q and enter Range'(p) at two distinct nodes. If any r satisfying the conditions above is structured with respect to predicate q, then by definition predicate p is structured with respect to q, a contradiction. Therefore, Unstruct(r,q). □

Theorem 3.7: If Struct(p1,p2) and there exists p3 such that Range'(p2) = Range'(p3), then Struct(p1,p3).

Proof: The conditions that resulted in Struct(p1,p2) must hold identically for p1 and p3 because the ranges of p2 and p3 are equal. □

Corollary: The same is true if Struct is replaced by Unstruct.

Theorem 3.8: If Range'(p) and Range'(q) are disjoint, then

$$\forall r \forall s\, [r \in \text{Range}'(p) \wedge s \in \text{Range}'(q) \Rightarrow \text{Struct}(r,s)].$$

Proof: If the ranges of p and q are disjoint, then so must be the ranges of predicate r in Range'(p) and predicate s in Range'(q). □

IV. PATHS

Although the nesting and structuredness properties developed in Chapter III largely characterize the control flow surrounding each node in a flowgraph, they do not provide a full characterization. Consider the control flow context of nodes x and y in the flowgraph of Figure IV-1. Both are nested in identical structured constructs; thus their PEN, NLP and UNST values are identical. But there is a difference that is not accounted for by the measures—the different ways in which the nodes can be reached from the start node. Such information can be valuable to a programmer who is focusing attention on one particular node and needs to know the set of nodes that could precede that node in any execution sequence. A control flow property that satisfies this need is the set of paths from the start node to the node of interest. A corresponding measure is the cardinality of the set of such paths.

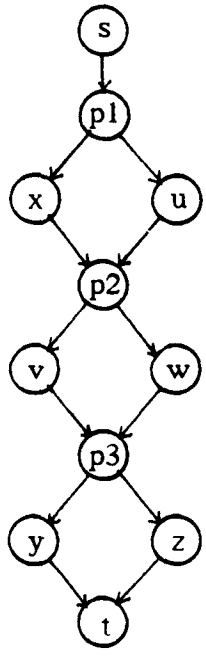


Figure IV-1. Flowgraph illustrating context difference

Although this property is intuitively appealing, the notion of "path" must be defined before it can be of any use. The logical definition for the set of paths is the set of all possible execution paths. But, for any program containing a loop, this set can be infinite, and infinite sets are impractical as bases for measures. Thus, the problem of defining the property becomes one of identifying a finite set of paths that adequately represent the possibly infinite set of execution sequences.

Section A addresses this question of adequacy by first defining "loop", and then by identifying several properties that an adequate set of paths should have. In Section B, several proposed finite sets of paths are evaluated against these properties, and are found to be deficient. Section C presents a definition for a set of paths that satisfies all but one of the properties (that property is shown to be unsatisfiable by any finite set of paths). However, the new property produces too much information to be of practical use to a programmer. Therefore, a fifth property, the predecessor set, is defined in Section D as an alternative to the path property. But, because the paths issue is important, Section E relates path sets and regular expressions as well as presenting bounds on the number of paths contained in specific types of flowgraphs.

A. Loops and Properties of Finite Sets of Paths

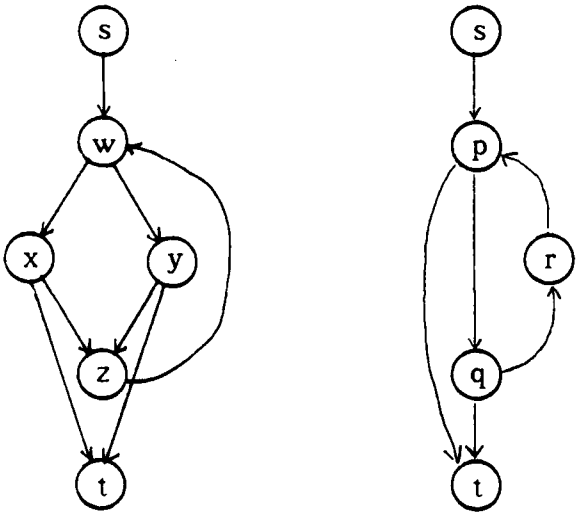
1. Loops

The aim of including the paths property in the set of characterizing properties is to provide the programmer usable information about the ways in which control can reach a certain node. One prime concern is that of adequately representing paths that contain loops; acyclic paths can represent themselves. Much of this problem lies in defining "loop". The concept of a program loop is intuitively understood by computer scientists as a programming language construct. However, the intuitive concept does not provide a good foundation upon which to build a path definition.

Viewing a loop as a single entity ignores the fact that many loops are comprised of more than one elementary cycle. Each distinct path through these loops must be

considered separately. Thus, it seems reasonable to define a loop as a set of cycles with a common predicate. However, this definition does not suffice. Consider the flowgraph in Figure IV-2(a). Node w predicates two cycles, w-x-z-w and w-y-z-w. Each cycle has a different exit arc. The first exits along arc (w,y) and the second along arc (w,x). The union of these two cycles would produce a loop in which the exit arcs are part of the loop, a seeming contradiction. Thus, the definition of loop must be restricted further. The cycles comprising a loop not only must be predicated by the same node, but also must have the same exit arc. This leads to the definition of loop used throughout the rest of this chapter. Elementary cycles are redefined to add some needed terminology.

Definition 4.1: For a flowgraph G = (N, E, s, t), node p ∈ N *predicates an elementary cycle* C if p ∈ C and there exists a path from an immediate successor q of p to t that does not contain any nodes in C. Arc (p,q) is called the *exit edge* of C, and q is called the *exit node.*



(a) (b)

Figure IV-2. Flowgraphs illustrating loops and predicates

Definition 4.2: A loop in flowgraph G predicated by node p with exit edge (p,n), denoted *Loop(p,n)*, is the set of all elementary cycles in G predicated by p that do not contain node n.

Looking again at the flowgraph in Figure IV-2(a), the two loops predicated by node w are Loop(w,x) = {w-y-z-w} and Loop(w,y) = {w-x-z-w}.

Sometimes an iterative construct can have more than one predicate and, therefore, more than one exit point. In this case, each predicate defines a distinct loop. Although the set of nodes in each loop is the same, the endpoints of the cycles, and, therefore, the orderings of the nodes differ. The flowgraph in Figure IV-2(b) has two predicates, p and q. The loop predicated by node p is p-q-r-p and the loop predicated by node q is q-r-p-q.

After identifying the set of loops in a flowgraph G, any execution path can be constructed by first selecting an appropriate acyclic path from s to t, then systematically substituting for each loop predicate on the path one or more of the cycles that comprise the loop for that predicate.

2. Path subset properties

Not all sets of paths can adequately represent the set of all execution paths. This section presents five desirable properties that a criterion for selecting a representative set should satisfy. In the following definitions and properties, EP(G) denotes the set of all possible execution paths in a flowgraph G. FS(G) denotes any finite subset of EP(G). All the paths in EP(G) are finite; they represent execution sequences from s to t in G of halting executions of G. Before defining the desired properties of subset criteria, the notion of a criterion is formally defined.

Definition 4.3: A *path subset criterion* is a boolean function C with domain $\{FS(G) \mid FS(G) \subseteq EP(G)\}$. C(FS(G)) is true if and only if FS(G) satisfies criterion C.

The reason for defining properties that subset criteria should satisfy is not to completely and rigidly specify the contents of a finite set of paths. Rather the properties are intended as guidelines, a yardstick against which individual criteria can be measured. The intent is to add formalism and rigor to what has often been a largely intuitive process.

The first property, one of immediate interest to software measures researchers, is that all finite subsets of EP(G) satisfying a particular criterion have the same cardinality. This alleviates the need to define an "optimal" size for a set of paths or to average many different set size values into one result. The first desirable property of path subset criteria is defined formally as:

> *Property 4.1:* For a path subset criterion C and a flowgraph G, if $\exists\, k\, [k \in \text{Integers} \wedge \forall\, FS(G)[C(FS(G)) \rightarrow |FS(G)| = k]]$, then C is a *fixed-size subset criterion.*

That criterion C yields fixed-size subsets does not imply there is only one subset FS(G) satisfying C. For instance, there can exist more than one basis set for a flowgraph G that satisfies the property that the cardinality of the set equals the cyclomatic number of G [20].

A second desirable property relates FS(G) to G. For any finite set of paths to adequately represent the control flow in a flowgraph, every edge in E must lie on some path in FS(G). Otherwise, FS(G) does not completely represent G. Thus, the second property is:

> *Property 4.2:* A finite subset FS(G) for a flowgraph G is *complete with respect to G* if and only if G can be reconstructed from FS(G). A path selection criterion C is *complete with respect to G* if and only if all subsets satisfying C are complete with respect to G.

This completeness property requires only that each edge in the flowgraph appear on at least one of the paths in FS(G). This property also implies that EP(G) can be

generated from FS(G) since the corresponding flowgraph can be generated from any complete subset, and from the flowgraph, EP(G) can be generated algorithmically.

As stated in Section 1, a major deficiency in existing path subset criteria is their failure to accurately represent loops in execution paths. Correcting this inadequacy requires that FS(G) reflect each possible path through a loop and each possible loop occurrence along an execution sequence. This leads to the third desirable property of path subset criteria.

> *Property 4.3:* For a flowgraph G, a finite subset FS(G) is *loop adequate with respect to G* if there exist paths in FS(G) that (1) represent a loop each time it can occur along an execution path and (2) represent each possible path through a loop at each place it can occur. A path subset criterion C is *loop adequate with respect to G* if all subsets satisfying C are loop adequate.

This property does not imply that there must exist one path in FS(G) that contains every possible loop occurrence and every possible path through a loop. It requires only that all of the information be obtainable directly from some set of paths in FS(G). One can argue that this information can be obtained from any complete finite subset, because the flowgraph G can be reconstructed from the subset, and all possible execution sequences generated from G. However, the intent of providing sets of paths to the programmer is to summarize possible execution paths through specific parts of the flowgraph, and to "unwind" loops along the paths to clearly expose all the possible ways of getting from the start node to some node of interest. Property 4.3 requires that the paths shown to the programmer provide a complete picture of iterative control flow.

A path subset criterion may establish a relation between the exhaustive set EP(G) and a finite subset FS(G). One such relation is that FS(G) be obtained from EP(G) by partitioning EP(G) and then selecting one path from each partition. This relation makes the finite subset in some sense representative of the exhaustive set.

Property 4.4: For a flowgraph G, FS(G) is *representative of* EP(G) if and only if (1) there exists a well-defined partitioning of EP(G) and (2) there exists a well-defined strategy for selecting one element from each partition to be included in FS(G). A path selection criterion is *representative* if and only if all subsets satisfying C are representative.

This property by itself is easy to obtain. Consider the case of partitioning EP(G) into sets of paths of length greater than or equal to some n and those of length less than n. FS(G) will consist of only two paths, but will yield little useful information about EP(G) or the underlying flowgraph G. The property takes on more significance if each path $p \in FS(G)$ can be used to recreate the entire partition from which p was obtained.

Property 4.5: If for a flowgraph G, FS(G) is representative of EP(G) and if each partition of EP(G) can be reconstructed from a path $q \in FS(G)$, then FS(G) is *reconstructive*. A path selection criterion C is *reconstructive* if and only if all subsets satisfying C are reconstructive.

Any reconstructive finite subset is necessarily loop adequate. However, there may exist loop adequate finite subsets which are not obtained from a partitioning of EP(G) and, hence, are not obviously reconstructive.

B. Analysis of Common Criteria for Finite Sets of Paths

In each of the following subsections a different path subset criterion is reviewed. Most of the criteria come from the published literature, although there are instances where the intended criterion is vaguely defined. In other instances, a criterion definition is clear, but analysis of the properties of the criterion is omitted. Again, the intent is not to imply that the set of properties presented in Section A is complete, but only to suggest that this type of more rigorous consideration is important.

1. Acyclic paths

Probably the most straightforward path subset criterion is for each element of FS(G) to be an acyclic execution path of G. Recall that an acyclic path is one on which no node occurs more than once. The criterion can be formally stated as:

Criterion C_1: Let G be a flowgraph and let AP(G) denote the set of all acyclic execution paths in G, then $C_1(FS(G)) \equiv FS(G) = AP(G)$.

Note that if flowgraph G is acyclic, then it is composed solely of acyclic paths, and EP(G) = AP(G).

This simple criterion satisfies two of the desired properties. First, the set of acyclic paths in each flowgraph G is unique, and, therefore, of fixed size. Thus, criterion C_1 is a fixed-size subset criterion. Secondly, EP(G) can be partitioned such that all the paths in each partition contain the same acyclic path. The contained acyclic path can be determined for any given path P by repeatedly removing all elementary cycles from P until none remain. When this is done, only the acyclic path remains. The acyclic path corresponding to each partition can be selected for FS(G). Thus, the criterion C_1 is representative.

Since none of the paths in FS(G) contain cycles, C_1, in general, is not complete with respect to G. For instance, suppose the program represented by flowgraph G is composed of a single *while* statement. Any path from s to t that contains control flow inside the *while* has to have two occurrences of the *while* predicate. But acyclic paths can contain only one occurrence of any node. Thus, no path in AP(G) can contain any of the control flow inside the loop. Therefore, the loop cannot be reconstructed from the paths in FS(G). By the same argument, C_1 is not loop adequate. Finally, since the paths containing loops in the partitions of EP(G) cannot be formed from FS(G), C_1 is not reconstructive.

2. Basis sets of execution paths

A *basis* set of execution paths for a flowgraph G is a set of linearly-independent paths from which any path through G can be reconstructed. Linearly independent means that none of the paths in the set can be constructed as a combination of others in the set. The best known basis is the one described by McCabe [2], reviewed in Chapter II. However, McCabe's basis is a set of cycles instead of paths. Baker [20] provided an algorithm for deriving a set of basis paths from s to t in G. His method is as follows:

```
(1) Construct a depth-first spanning tree T for G.
(2) Place the path from s to t in T into the basis set B.
(3) For each edge e in E that is not in T do
      Add e to T;
      Select a path from s to t in T containing e,
        and add the path to B;
    endfor
```

Baker showed that while B may not be unique, because in general a flowgraph has many depth-first spanning trees, the cardinality of B equals the cyclomatic complexity of G. This basis satisfies the following criterion.

Criterion C_2: Let G be a flowgraph with cyclomatic number V(G). Then, C_2 is a path subset criterion where $C_2(FS(G)) \equiv |FS(G)| = V(G)$ and the paths in FS(G) are linearly independent.

Since all of the FS(G) satisfying C_2 must contain exactly V(G) many paths, C_2 is clearly a fixed-size subset criterion. That C_2 is complete follows immediately from the the definition of basis (see the first paragraph in this section).

C_2 is not, in general, loop adequate. Each path in FS(G) may contain no more than one occurrence of one cycle of a loop. All possible loop occurrences along some given execution path may not be reflected.

The construction of the basis set B suggests a partitioning of EP(G). Each path in B contains one arc that does not occur in any other path in B (otherwise, the paths

would not be linearly independent). EP(G) can be partitioned according to the unique arc in each path in B. There will be V(G) partitions. For any path P in EP(G), if P contains just one of the "unique" arcs, then it is placed in that arc's corresponding partition. However, if P contains two or more unique arcs, the problem arises of selecting a partition to place it in. The choice cannot be arbitrary, because then the partitioning would not be well-defined. A solution is to number the unique arcs in the order in which they are added to the spanning tree when creating B. Then, if a path contains more than one unique arc, the path can be placed in the partition corresponding to its lowest numbered unique arc. Then, FS(G) can be constructed by selecting from each partition the basic path corresponding to that partition. Thus, C_2 is representative.

Clearly, though, there can be paths in a partition that contain arcs not in the partition's representative path. Thus, there can be paths in each partition that cannot be generated from the corresponding path in FS(G). Thus, C_2 is not reconstructive.

3. No repeated cycles

A set of paths on which no loop occurs more than once is an often used criterion in measures research [8, 12]. Unfortunately, none of the uses of this criterion defines exactly what is meant by a "loop", whether it includes all the cycles comprising the loop, or just a single cycle. To allow a reasonably rigorous definition of this criterion, the assumption will be that a loop refers to all the cycles. Thus, the criterion allows all paths containing no more than one instance of each elementary cycle in the flowgraph. Formally, this criterion is:

Criterion C_3: Let G be a flowgraph and let EC(G) be the set of elementary cycles in G. Path subset criterion C_3 is

$$C_3(FS(G)) \equiv FS(G) = \{P \mid P \in EP(G) \wedge$$
$$\forall c\, [(c \in EC(G) \wedge c \text{ is a subpath of } P)$$
$$\Rightarrow c \text{ appears only once on } P]\}.$$

Since there is exactly one subset of EP(G) that satisfies this criterion, C_3 is a fixed-size subset criterion. The FS(G) satisfying C_3 will contain all the acyclic paths in G as well as every elementary cycle, because each elementary cycle must lie on at least one path in G. Thus, FS(G) contains all the information needed to construct G, and is, therefore, complete with respect to G.

C_3 is not loop adequate. Consider the flowgraph in Figure IV-3. One of the possible execution paths through G is s-a-b-b-c-a-b-b-c-t. The FS(G) satisfying C_3 does not contain a path on which the b-b elementary cycle appears both before and during the execution of the outer loop. To accurately reflect the loops, FS(G) must generate this type of path. Since it cannot, C_3 is not loop adequate.

C_3 is not representative, although it does suggest a finer partition of EP(G) than do the earlier criteria. Consider the partitioning of EP(G) such that each path is placed in a partition according to the first cycle that appears on the path. Each acyclic path will appear in a partition with exactly one member. While this is a finer partitioning, an FS(G) satisfying C_3 cannot be produced from the partitions. In

FS(G): {s-a-b-c-t,
s-a-b-b-c-t,
s-a-b-c-a-b-c-t,
s-a-b-b-c-a-b-c-t,
s-a-b-c-a-b-b-c-t}

G:

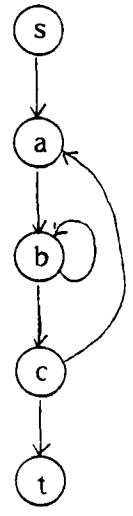


Figure IV-3. A finite subset that is not loop adequate

general, there will be many distinct paths in FS(G) with the same first cycle. Finally, since there appears to be no plausible partition that makes C_3 representative, it is not reconstructive.

4. Paige's criterion

The finite set of paths satisfying criterion C_3 directly provides more information about the control flow in a program than does the basis set of execution paths satisfying criterion C_2. However, no tractable method of generating the set satisfying C_3 has been published. But, Paige describes a finite set of execution paths that provides as much information and that can be generated tractably [40, 41].

Paige constructs his set by first including all the acyclic paths in a flowgraph G. He creates new paths by adding to each of the acyclic paths subpaths that begin and end with nodes on the acyclic paths and that contain no other nodes lying on the acyclic paths. These new paths are added to his set. He repeats the previous step with the new paths until all subpaths have been added to the set.

This process can be formally described by the definitions (from [40]) and the algorithm, below.

Definition 4.4: A *level-0* path is an acyclic path from s to t in G.

Definition 4.5: A *level-i* path, $i > 0$, is a path $x_1 - x_2 - \cdots - x_n$ such that (1) x_1 and x_n are nodes on a level-k path where $k < i$, and (2) none of the nodes $x_2, x_3, \ldots, x_{n-1}$ appear on any level-k path for $k < i$.

Note that level-i paths are not, themselves, execution paths from s to t.

Criterion C_4: A finite set FS(G) satisfies criterion C_4 if it is constructed as follows:

```
(Assume a maximum of n levels of paths.)
FS(G) = { P | P is a level-0 path in G};
For i := 1 to n do
  For each level-i path R in G do
    (* Assume R is a path from node x to node y *)
```

```
For each path P in FS(G) do
  If nodes x and y lie on P then
    let P1 be the part of P from s
       to the immediate predecessor of x;
    let P2 be the part of P from the
       immediate successor of y to t;
    construct a new path P' = P1RP2;
    Add P' to FS(G)
  endif
 endfor
endfor
endfor.
```

Paige proved that the set of level-i, $i \geq 0$, paths in G is unique. Therefore, the FS(G) constructed above is unique, and C_4 is a fixed-size subset criterion. Since each node and arc in G are contained in some path in FS(G), G can be reconstructed from FS(G); therefore, C_4 is complete.

C_4 is not loop adequate for the same reasons C_3 is not loop adequate. The level-i paths represent cycles or parts of cycles in G. Since each cycle can appear only once on each path in FS(G), paths on which cycles occur, nonconsecutively, two or more times cannot be generated from any path in FS(G). C_4 is also neither representative nor reconstructive for the same reasons as C_3.

Paige pointed out in [40] that there are cases that are not handled in an intuitively appealing manner by his criterion. Consider the flowgraph of Figure IV-4, which Paige called the "nemesis flowgraph". It is an instance of Hecht's irreducible (by intervals) "*" subflowgraph [13] with multiple exit points. Using Paige's criterion, only level-0 paths are obtained, although this flowgraph is commonly viewed as containing a multiple-entry, multiple-exit loop. The finite subset generated by C_4 does not reflect any elementary cycle in the flowgraph.

5. Summary

Table IV-1 summarizes the properties satisfied by the four reviewed criteria. It shows that none of the criteria are reconstructive; however, as shown in the next section, no criterion can be reconstructive. More importantly, the table reveals that

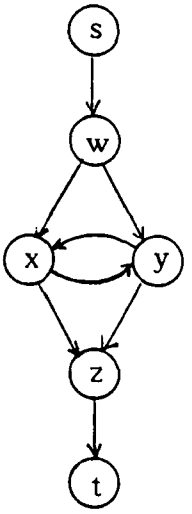


Figure IV-4. Paige's "nemesis flowgraph"

none of the most common subset criteria adequately represent all the possible occurrences of iteration on execution sequences. A major reason for this, again, appears to be the lack of a rigorous definition for loops in programs. However, the definition of a loop given in Section A forms the basis for a criterion that does satisfy the loop adequacy property. That criterion is presented in the next section.

Table IV-1. Summary of properties satisfied by subset criteria

Criterion	Property				
	fixed size	complete	loop* adequate	representative	reconstructive
Acyclic Paths	yes	no	no	yes	no
Basis Sets	yes	yes	no	yes	no
No Repeated Cycles	yes	yes	no	no	no
Paige's	yes	yes	no	no	no

C. An Alternate Criterion

One deficiency common to the criteria reviewed in the previous section is the lack of adequate loop representation. To provide an accurate representation, a criterion must select a path set that (1) shows every possible combination of loop occurrences on any possible execution path, and (2) shows all the possible paths through each loop each time the loop can occur. However, this requirement alone would select almost all of EP(G). To keep the set size reasonable, each occurrence of a loop on a path could be represented by one iteration of one cycle of that loop. To show that the loop is comprised of n cycles, the set could contain n paths that are identical except for the cycle chosen to represent the loop at that point, plus one more path containing none of the cycles to reflect no iteration at the same point. Multiple consecutive iterations of one or more of the cycles comprising a loop would not provide any additional information, and would not be included in the finite set. These observations and desired properties lead to the following new criterion:

> *Criterion* C_5: $C_5(FS(G)) \equiv FS(G)$ = the set of all paths from s to t in flowgraph G such that (1) no path contains two or more consecutive occurrences of cycles from Loop(p,q) for all loop-predicate, exit-node pairs (p,q) in G, and (2) if a node p predicates $n > 1$ loops, then no path contains more than n consecutive occurrences of cycles predicated by p.

Condition 1 restricts the representation of a loop at a given point to just one iteration of one cycle in the loop at that point. The need for the second condition is illustrated by the flowgraph in Figure IV-5. By definition, node x predicates Loop(x,y) = {x-z-x} and Loop(x,z) = {x-y-x}. Condition 1 allows paths of the form s-x-y-x-z-x-y-x-z-x-...-t. Thus, condition 1 alone does not even define a finite set of paths. This happens because nodes y and z are in distinct loops and are exit nodes for the loop in which they do not lie. Condition 2 corrects this problem by limiting the number of consecutive iterations of two loops predicated by the same predicate to

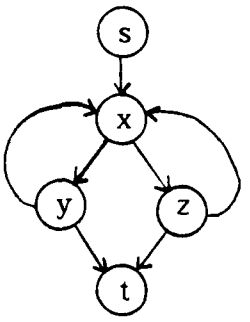


Figure IV-5. Distinct loops with exit nodes in other loops

two, the number of loops.

The criterion does not restrict the points on a path at which a loop can occur, nor does it restrict the number of nonconsecutive occurrences of a loop on a path. Therefore, any FS(G) satisfying C_5 will contain paths that reflect all possible combinations of loop occurrences on paths, and that show all possible paths through a loop at each place the loop can occur. Thus, the criterion is loop adequate. It also satisfies three other properties; it is not reconstructive.

Only one set of paths in any flowgraph satisfies the criterion, thus, it is a fixed-size criterion. The paths in FS(G) contain every arc in G, so flowgraph G can be reconstructed from FS(G), thus, the criterion is complete. For a flowgraph G, EP(G) can be partitioned based on the cycle traversed on the first iteration of each loop occurrence on an execution path. Each acyclic path forms its own distinct partition. Then, those paths with only one iteration of each loop can be chosen as elements of FS(G). Thus, the criterion is representative.

The criterion is not reconstructive. All of the many possible ways of iterating multiple-cycle loops cannot be generated from a single path. Consider all the possible ways of iterating a loop comprised of two or more cycles. No matter how fine the infinite set is partitioned into non-trivial partitions, and no matter how the representative path from each partition is selected, there will always be an execution

path that cannot be generated directly from the finite set. Thus, for any flowgraph containing multiple-cycle loops, the only criterion that will satisfy all the properties is $C(FS(G)) \equiv FS(G) = EP(G)$. But then FS(G) is no longer a finite set if G contains loops. Therefore, no criterion exists that will satisfy all the desired properties for all flowgraphs.

Baker, Howatt and Bieman [42] describe an alternative method for defining Criterion C_5. They use a representation of a flowgraph G called the *path tree.* A path tree is a structure built to represent all the acyclic paths, all the elementary cycles that comprise a loop, and all possible combinations of loops in G. Each node in the tree except the root represents a path in G. The root node is the start node of G. The children of the root are the acyclic paths from immediate successors of the start node to t. The remainder of the tree nodes are paths that represent elementary cycles in G. To retain the association between loop predicates and the cycles they predicate, the tree nodes that represent elementary cycles are considered children of their predicate node in the parent tree node, instead of children of the parent node itself.

The path tree is formally defined below, followed by an example to illustrate its construction.

Definition 4.6: A path tree T for a flowgraph G is a tree that is constructed by the following algorithm: (Assume that all the loop predicates, exit nodes, and their associated loops in G have been identified.)

(1) Create the root of T and label it s;

(2) For each acyclic path P in G
    from the immediate successors of s to t do
      Add P as a child of s;
      Mark P unprocessed
    endfor

(3) Repeat
      Let R be an unprocessed tree node in T;
      Mark R as processed;
      For each loop predicate graph node q in R
        such that graph node n immediately follows q in R do

```
        For each cycle C in Loop(q,n) in G do
          Add tail(C) as a child of q in R in T;
          Mark C unprocessed
        endfor
      endfor
    Until all leaves in T are marked processed;
```

Note: tail(C) is C without the first node.

To illustrate this algorithm, a path tree for the flowgraph shown in Figure IV-5 is constructed. Its loops are: Loop(x,y) = {x-z-x}, Loop(x,z) = {x-y-x}, Loop(y,t) = {y-x-y}, and Loop(z,t) = {z-x-z}. The acyclic paths in G are s-x-y-t and s-x-z-t; thus, after step two of the algorithm, the tree T looks as shown in Figure IV-6.

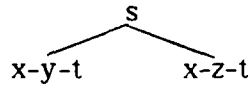


Figure IV-6. Path tree after adding acyclic paths

Since neither leaf has been processed, select x-y-t. There exists a Loop(x,y) in G, so tail(x-z-x) is added to T as the child of x. There also exists a Loop(y,t) in G, thus tail(y-x-y) is added as a child of y. The tree now looks as shown in Figure IV-7.

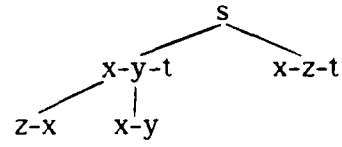


Figure IV-7. Path tree after adding two cycles

To complete the left side of the tree, the two left leaves are processed. There is no Loop(z,x) in G; the tree node z-x is a leaf of the path tree. There is a Loop(x,y), so z-x gets added as a child of node x in the tree node x-y. Since there is no Loop(z,x) in G, the left side of the tree is complete, as shown in Figure IV-8.

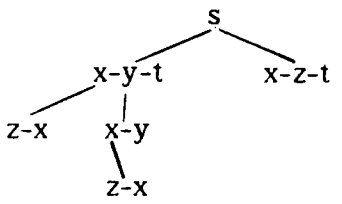


Figure IV-8. Path tree with complete left half

The remaining unprocessed child of s is processed similarly, completing the path tree, shown in Figure IV-9.

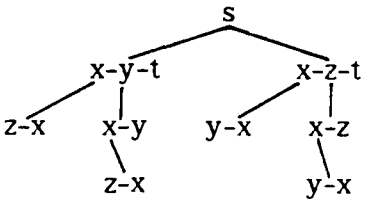


Figure IV-9. Complete path tree

The tree reflects the looping structure of the flowgraph G. If a path with the underlying acyclic path s-x-z-t is followed, upon reaching node x, the cycle x-y-x can be iterated. Then, upon reaching node z, the cycle z-x-z can be iterated. But within this cycle, the cycle x-y-x can again be iterated. Thus, one possible path through G is s-x-y-x-z-x-y-x-z-t.

The desired finite path subset criterion can be defined in terms of the path tree, but enumerating paths directly from the tree is complicated. Reflecting all possible combinations of iterations and noniterations of the cycles represented by nodes of the tree requires a very involved algorithm. A simpler approach is to build a lattice from the path tree such that every path from s to t in the lattice satisfies the desired criterion, and all paths satisfying the criterion are contained in the lattice.

An algorithm for constructing the path lattice is given in Figure IV-11. The abstract data types used by the algorithm are declared in Figure IV-10. Associated

```
Flowgraph = 4-tuple <Nodes, Arcs, S, T> where
    Nodes: set of GraphNodeIdentifier;
    Arcs : set of ordered pairs <From, To> where
              From, To: GraphNodeIdentifier
    S, T : GraphNodeIdentifier

PathTree = 3-tuple <Nodes, Arcs, Root> where
    Nodes: set of PathTreeNodeType where
           PathTreeNodeType: Sequence of GraphNodeIdentifier
    Arcs : set of ordered pairs <From, To> where
           From: ordered pair <PathTreeNode, GraphNode> where
                PathTreeNode: PathTreeNodeType
                GraphNode: GraphNodeIdentifier
           To  : PathTreeNodeType
    Root: PathTreeNodeType

PathLattice = 4-tuple <Nodes, Arcs, Top, Bottom> where
   Nodes: set of LatticeNodeType where
        LatticeNodeType = ordered pair <GraphNode, LatticeNodeID> where
                GraphNode: GraphNodeIdentifier
                LatticeNodeID: LatticeNodeIDType
   Arcs: set of ordered pairs <From, To> where
       From, To: LatticeNodeType
   Top, Bottom: LatticeNodeType
```

Figure IV-10. Type declarations

with some of the data types are functions that provide the means to access those types. Given a sequence of graph node identifiers $s = x_1x_2 \cdots x_{k-1}x_k$, $first(s) = x_1$, $last(s) = x_k$, $header(s) = x_1x_2 \cdots x_{k-1}$, $trailer(s) = x_2x_3 \cdots x_k$, $next(s,x_i) = x_{i+1}$ for $1 \leqslant i < k$, and $next(s,x_k) = \epsilon$. Because two or more lattice nodes can represent the same graph node, each lattice node contains a unique identifier supplied by the function NewLattNodeID.

In the algorithm, tuples are represented using the delimiters "<" and " >". Components of tuples are referenced using a functional notation; the root of a path tree T is referenced by Root(T).

An alternate form of the criterion can be defined in terms of the lattice.

Criterion C_5: $C_5(FS(G)) \equiv FS(G)$ is the set of all paths from s to t in the lattice built by the procedure of Figure IV-11.

```
Procedure BuildPathLattice (T: PathTree; var L: PathLattice);
var  L1, L2: LatticeNodeType;
     C: PathTreeNodeType;

     procedure SubLattice (var L: PathLattice;  Top, Bottom: LatticeNodeType;
                           P: PathTreeNodeType);
     var  L1, L2:  LatticeNodeType;
          N, FN: GraphNodeIdentifier;
          C: PathTreeNodeType;
     begin
       L1 := <first(P),NewLattNodeID()>;
       Nodes(L) := Nodes(L) ∪ {L1};
       Arcs(L) := Arcs(L) ∪ {<Top,L1>};
       for each graph node N in P do
         FN := next (P,N);
         if FN ≠ ϵ then
           L2 := <FN, NewLattNodeID()>;
           Nodes(L) := Nodes(L) ∪ {L2};
         else
           L2 := Bottom;
         Arcs(L) := Arcs(L) ∪ {<L1,L2>};
         for each tree node C such that <<P,N>,C> ∈ Arcs(T) do
           SubLattice (L, L1, L2, C);
         endfor;
         L1 := L2
       endfor
     end;

begin
  L1 := <first(Root(T)), NewLattNodeID()>;
  let C be any child of Root(T);
  L2 := <last(C), NewLattNodeID()>;
  Nodes(L) := {L1, L2};
  Arcs(L) := ∅;
  for each node C such that <<Root(T), first(Root(T))>,C> ∈ Arcs(T) do
    SubLattice (L, L1, L2, header(C));
end;
```

Figure IV-11. Algorithm to build a path lattice

The lattice output by this algorithm for the path tree in Figure IV-9 is shown in Figure IV-12. The paths, derived from this lattice, satisfying C_5 are:

s-x-y-t	s-x-z-t
s-x-z-x-y-t	s-x-y-x-z-t
s-x-y-x-y-t	s-x-z-x-z-t
s-x-y-x-z-x-y-t	s-x-z-x-y-x-z-t
s-x-z-x-y-x-y-t	s-x-y-x-z-x-z-t
s-x-z-x-y-x-z-x-y-t	s-x-y-x-z-x-y-x-z-t

Although the last two paths appear to contain three consecutive iterations of the

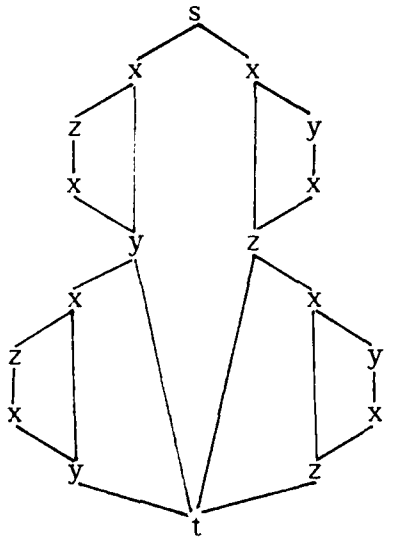


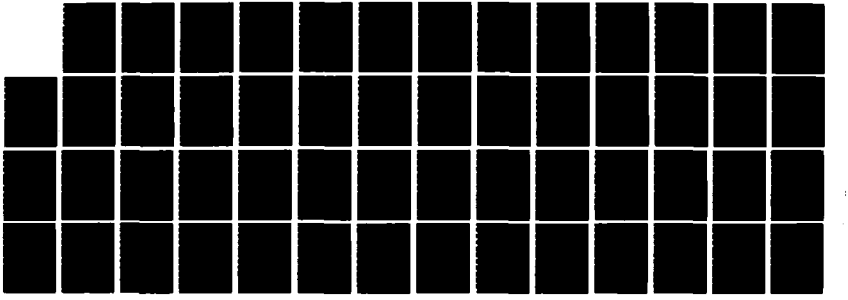
Figure IV-12. Path lattice for the path tree of Figure IV-9

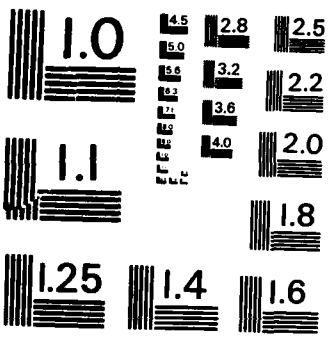
loops predicated by x, only the first and the third x-λ-x subsequences are actually loops on x. The second x-y-x in the path on the right is contained in an iteration of the loop predicated by z; the second x-z-x in the path on the left is contained in an iteration of the loop predicated by y.

The criterion itself provides no insight into generating a path count, but computing the number of paths from the path tree is reasonably straightforward. Each of the acyclic-path children of the root contributes one to the total path count. If a loop predicate p falls on one of the paths, then the count is increased by one plus the number of possible paths from p to p. The possible paths from p to p are represented as the descendants of p in the tree. If two or more loop predicates fall in a given path in a tree node, then the cycles they predicate fall in series in the flowgraph; the number of paths contained in the loop of each predicate are multiplied to obtain the total paths. This path computation is performed by the procedure in Figure IV-13.

AD-A166 762 A QUANTITATIVE CHARACTERIZATION OF CONTROL FLOW CONTEXT: SOFTWARE MEASUEE.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH J W HOWATT 1985 2/2

UNCLASSIFIED AFIT/CI/NR-86-37D F/G 9/2 NL





```
procedure ComputePaths (T: PathTree; var NumPaths: integer);
 var C: PathTreeNodeType;

 function PathCount (X: PathTreeNodeType): integer;
  var R: GraphNodeIdentifier;
     Y: PathTreeNodeType;
     M,N: integer;
  begin
   N := 1;
   for each graph node identifier R in X do
    if ∃q [<<X,R>,q> ∈ Arcs(T)] then
     M := 1;
     for each Y such that <<X,R>,Y> ∈ Arcs(T) do
      M := M + PathCount(Y)
     endfor;
     N := N * M;
    endif
   endfor
   return (N)
  end;

 begin  (* ComputePaths *)
  NumPaths := 0;
  for each C such that <<Root(T),first(Root(T))>,C> ∈ Arcs(T) do
   NumPaths := NumPaths + PathCount (C)
  endfor
 end;
```

Figure IV-13. Path computation algorithm

But the total number of paths through a flowgraph is not the desired measure. The philosophy motivating this research is to compute measures at the node level; thus, the appropriate paths to count are those from the start node to some given node x. Not only will the path count tell the programmer how many possible ways there are for control to reach node x, but the set of paths will also identify all the possible predecessors of x.

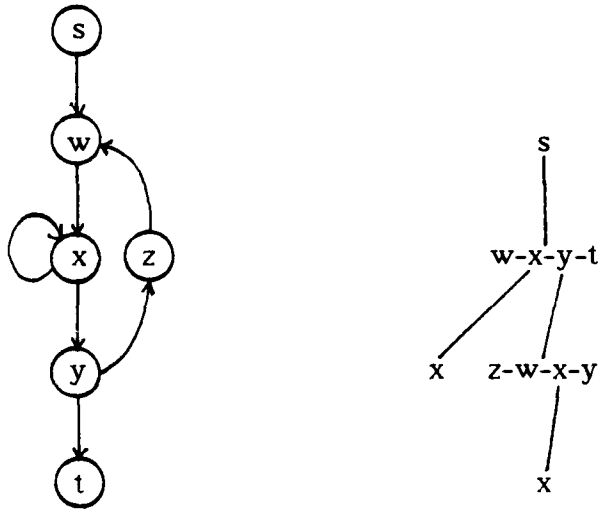
This path set should also satisfy the intent of criterion C_5. All cycles leading to and containing node n should be reflected, but in the same manner restricted by the criterion. One type of path that could be used is the first occurrence path, defined in Chapter III. Schneidewind and Hoffman used this path type in their reachability measure [16]. But consider the flowgraph in Figure IV-14(a). The set of first occurrence paths from s to w contains just one path. It contains no paths from s to w

through w. These paths are needed because they identify all the possible predecessors of node w. Thus, paths s-w-x-y-z-w and s-w-x-x-y-z-w must also be included to ensure that all the predecessors of w and the paths (and the loops on those paths) containing them are reflected.

A second method for generating the set of paths is to enumerate all the paths from s to a given node as represented by the path tree. However, that set does not necessarily include all the desired paths, either. Consider, again, the flowgraph in Figure IV-14(a) and its path tree in Figure IV-14(b). The set of paths from s to x, enumerated directly from the tree are:

s-w-x	s-w-x-x
s-w-x-y-z-w-x	s-w-x-x-y-z-w-x
s-w-x-y-z-w-x-x	s-w-x-x-y-z-w-x-x

This set is complete because it reflects that x can precede itself both before and during an iteration of the outer loop.



(a) (b)

Figure IV-14. Flowgraph and corresponding path tree

But now consider the paths from s to z that can be generated from the path tree. They are s-w-x-y-z and s-w-x-x-y-z. That z is a possible predecessor of itself (because of the outer loop) is not reflected in these paths. To show this, some of the paths must contain the cycles in the loop predicated by y, even though the exit node of y's loop does not appear. Substituting the cycle predicated by y for y on the paths yields the following additional paths:

s-w-x-y-z-w-x-y-z s-w-x-x-y-z-w-x-y-z
s-w-x-y-z-w-x-x-y-z s-w-x-x-y-z-w-x-x-y-z

Combining these paths with the two original ones yields a path set that does contain all the possible predecessors of z and that also satisfies the intent of Criterion C_5. All possible occurrences of each loop are reflected and no cycle is iterated twice in succession.

In general, if the node of interest lies in a loop and precedes the loop predicate on every path in which both occur, then all the cycles of that loop and all the predecessors of the node of interest will be reflected in the paths generated directly from the path tree. But if the node of interest, x, follows the first occurrence of a node that predicates a loop containing x, then the paths generated directly from the path tree will not include all desired paths. The cycles predicated by one or more of the nodes in LPred(x) will not be represented on the paths and will have to be added at the point where their predicates occur. The following definition describes how the path lattice can be modified to generate the desired set of paths.

Definition 4.7: Given a flowgraph G = (N, E, s, t) and any node n ∈ N, the *predecessor path set* of n, a finite set of possible execution paths from s to n in G, are contained in the lattice created by the following algorithm. Assume the path lattice L for G and LPred(n) have been computed.

(1) Remove all nodes and arcs that do not lie on any path from s to any occurrence of n.

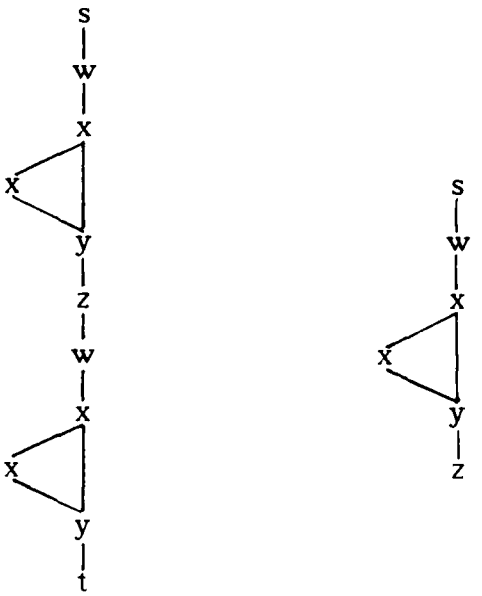
(2) For each occurrence in L of y ∈ LPred(n) do

if n is not both a predecessor and a successor of y and
y is not a successor of itself then
add paths from y to its immediate successor that
represent cycles in G predicated by y.

(3) For each predicate-node, exit-node pair (p,q)
on paths newly added in steps 2 and 3 do
add paths from p to q that represent the cycles in Loop(p,q).

The *if* statement in step 2 ensures that the loop predicated by y does not get included when it is not needed. If node n both precedes and follows node y, then the cycle containing n has already been included. If a copy of y follows itself, then the loop predicated by y has been included.

To illustrate how this algorithm works, the lattice for the paths from s to z in the flowgraph of Figure IV-14(a) will be constructed. The lattice containing all the paths from s to t is shown in Figure IV-15(a), and the reduced lattice produced by step 1 of the algorithm is shown in Figure IV-15(b). Node y is a member of



(a) (b)

Figure IV-15. Path lattices for the flowgraph of Figure IV-14(a)

LPred(z). Because y is not preceded by z in the reduced lattice, y's loop, the elementary cycle y-z-w-x-y, is added between nodes y and z to produce the lattice shown in Figure IV-16(a). Since y is the only node in LPred(z), step 2 is complete. Step 3 determines that Loop(x,y) must be represented in the path that was just added to the lattice, so the cycle x-x is added between x and y to complete the lattice, as shown in Figure IV-16(b). The paths represented by this lattice are:

s-w-x-y-z	s-w-x-x-y-z
s-w-x-y-z-w-x-y-z	s-w-x-x-y-z-w-x-y-z
s-w-x-y-z-w-x-x-y-z	s-w-x-x-y-z-w-x-x-y-z

These paths contain all the predecessors of z, and they reflect all occurrences of loops, without multiple consecutive iterations of the cycles in any loop. Thus, the paths satisfy the intent of Criterion C_5.

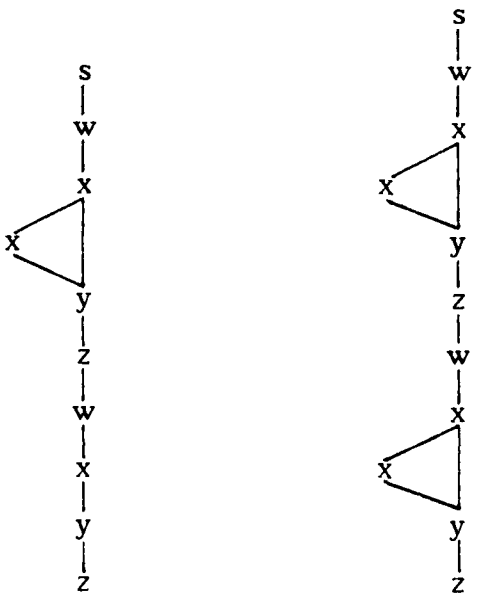


Figure IV-16. Lattices for paths from s to z

The set of all paths generated from a lattice constructed using the algorithm of Definition 4.7 will satisfy the intent of Criterion C_5. Step 1 of the algorithm only deletes parts of the lattice, and, therefore, creates no new representations of cycles. Step 2 considers separately each occurrence of a predicate y in LPred(n). If n does not precede y and y does not succeed itself, then there can be no iterations of any of y's cycles along the paths containing that occurrence of y. Adding parallel paths to represent the cycles predicated by y cannot introduce consecutive iterations of any of the cycles. Step 3 adds paths representing cycles in nested loops where a loop-predicate, exit-node pair occurs on a newly added path. Since an exit node for a containing loop cannot occur in a nested loop, and since loop-predicate, exit-node pairs for the same loop cannot occur in succession, no loop can be represented twice in succession along the same path. Thus, the lattice will not contain consecutive iterations of cycles in the same loop, nor more than n consecutive cycles from n loops predicated by the same node.

All possible loops will be represented in the set. Step 2 adds paths representing cycles to reflect all the possible paths through a loop containing n. Step 3 adds paths everywhere they are needed to represent loops nested in the ones added in step 2.

Since loops are reflected everywhere they can occur, but are represented by only single iterations of their constituent cycles, the paths in the lattice satisfy the intent of Criterion C_5.

Once the lattice is complete, the set of paths from s to a given flowgraph node x can be generated by a simple depth first search of the lattice. Each time the node x is encountered, the path followed to reach x can be added to the set. The measure, the number of paths from s to x in the flowgraph, can be computed by counting the paths in the set.

D. Paths Reconsidered

As will be shown in the next section, the number of paths through a flowgraph that satisfy Criterion C_5 can be more than exponential in the number of predicates. Providing a programmer an enumeration of so many paths seems, at a practical level, to be little better than providing no path information at all. This also applies to computing a six-figure measure value. With values that large, a difference of one or two orders of magnitude in numbers of paths will not seem significant. Thus, a path count is not as practical a measure as it may seem.

Throughout the development of the paths measure, one criterion that had to be satisfied was that the paths from s to a given node x contain all the possible predecessors of x. Therefore, it seems reasonable that if the paths measure is not practical, then a count of predecessor nodes can be used instead. Considering the flowgraph of Figure IV-1 again, a count of the predecessors of nodes x and y distinguishes their relative positions in the flowgraph as well as a count of paths from s to the nodes. And, enumerating the predecessor set for each node would not overwhelm a programmer as enumerating a set of paths may. Therefore, the fourth property for the set of properties chosen to characterize control flow is the set of predecessors of a given node.

Definition 4.8: Given a flowgraph $G = (N, E, s, t)$, the set of nodes that precede any occurrence of a given node $n \in N$ on any path from s to n, referred to as the antecedents of n, Ante(N), is

$$Ante(n) = \{x \mid x \in N \wedge \exists P [\, P \text{ is a path from s to n} \wedge x \in P \,]\}.$$

The measure that reflects this property is defined as follows.

Definition 4.9: Given a flowgraph $G = (N, E, s, t)$,

$$\forall n [\, n \in N \Rightarrow NAnte(n) = Cardinality(Ante(n))].$$

Chapter V presents an algorithm for computing Ante and NAnte, and illustrates their utility.

The information provided in Ante tells a programmer not all the different ways of reaching a given node n, but all of the possible statements that can be executed prior to an execution of basic block n. This information can be more easily digested, and is, therefore, of more use. Both the property and its measure are far easier to compute than the set of paths. Preliminary investigations of implementing path algorithms revealed an expected time complexity of $O(n^4)$. The predecessor set can be computed easily in time $O(n^2)$.

This argument for the predecessors property does not infer that the paths property should be ignored. It may prove useful in characterizing control flow and will probably be useful in other research areas such as test strategy development and automatic test case generation. The predecessors measure was selected because one of the prime objectives of this research is to provide properties and measures that can be of immediate use to a programmer, regardless of their merit as complexity measures. The Ante property satisfies that objective more so than the paths property, because of the time needed to generate the paths and because the Ante information is less likely to overwhelm a programmer.

E. Further Observations on the Path Criterion

This section discusses the relationship between regular expressions and criterion C_5. It also presents an interesting observation on the difference in the number of paths generated by *while* and *repeat* constructs. Regular expressions are interesting because they represent a compact method for storing all of the control flow information for a program. Pattern matching tools could be used on regular expressions to analyze the control structure. The second set of observations, on bounds for numbers of paths in programs, provides information for researchers in testing methodologies. Specifically, it provides discouraging news to proponents of the "test all paths" school of testing.

1. Regular expressions

The original motivation behind the development of the path tree was that of finding a method to produce the set of paths that can be generated from a regular expression for a flowgraph when the closure operator "*" is taken to mean "repeat zero or one times only". Regular expressions themselves were not used for two reasons. First, no method had been defined for producing minimal regular expressions from flowgraphs. "Minimal" does not necessarily mean the shortest, but implies that the expression contains no redundant subexpressions such as $(\alpha^*)^*$, $\alpha + \beta + \alpha$, and $(\alpha + \epsilon)^*$, where α and β are strings of node identifiers, and ϵ denotes the empty string. An expression such as *(abdf+abef)* is considered minimal, even though it can be written more compactly as *ab(d+e)f*, because it does not generate a redundant string. There is no way to generate two identical strings by using two distinct subexpressions in the regular expression.

Secondly, some minimal regular expressions are too concise. They do not accurately reflect loop nesting as required by Criterion C_5. For example, the regular expression $sx(y+z)(x(y+z))^*t$ reflects the control structure of the flowgraph in Figure IV-5. However, it does not accurately reflect that the flowgraph contains four loops. To show all the loops and how they could be nested, the regular expression that should be used is $s(x(zx)^*y(x(zx)^*y)^*+x(yx)^*z(x(yx)^*z)^*)t$.

Interestingly, the method of building the path tree suggests a way to generate the desired regular expression. The construction is similar to that of the path tree, but more deterministic. Let the alphabet Σ be the set of node identifiers for a flowgraph G = (N, E, s, t). The algorithm first forms the union of all the acyclic paths in G and then inserts into each acyclic path the cycles predicated by predicate-node, exit-node pairs lying on the path. Each of the inserted cycles must in turn be scanned so nested cycles can be added. The procedure in Figure IV-17 implements this algorithm, generating a regular expression from a flowgraph. As with the path

tree construction algorithm, all loop predicates, exit nodes, and their corresponding loops are assumed to be available.

The minimality of the constructed regular expression can be argued from the algorithm. Each acyclic path is added exactly once. Since each is unique, no redundancies are introduced. Each predicate-node, exit-node pair can occur no more than one time per acyclic path; thus, a loop will be added exactly once on any acyclic path. The same argument applies to nested cycles added to loops. Each cycle in a loop is unique, and a predicate-node, exit-node pair can occur at most once per cycle. Therefore, no redundancies can be added by inserting nested loops. Finally, the

```
procedure ConstructRE (G: Flowgraph);
var P: PathType;

  procedure GenerateSubstring (P: path);
  var M, N: NodeType;
     Q: PathType;
 begin
   for each node N in P from left to right do
     print (N);
     let M be the immediate successor of N in P;
     if Loop(N,M) exists then
       print ("(");
       for each cycle Q in Loop(N,M) do
         GenerateSubstring (tail(Q));
         if there are more cycles to process then
           print ("+");
       endfor;
       print ( ")*" )
     endif
   endfor
 end; (* Generate Substring *)

begin (* ConstructRE *)
  print ( "s(" );
  for each acyclic path P in G do
    GenerateSubstring (tail(P));
    if there are more acyclic paths to process then
      print ( "+" );
  endfor
  print ( ")" )
end;
```

Figure IV-17. Procedure to construct regular expression

algorithm will halt. The number of acyclic paths is finite, as is the number of cycles comprising any loop. No loop will be inserted nested within itself because no nested loop can contain an outer loop's exit node. The recursion will stop when the innermost nested loop has been inserted. Thus, the regular expression produced by the procedure is finite and minimal.

The number of paths satisfying criterion C_5 can be computed directly from a regular expression constructed by the algorithm above. A grammar for doing this is presented in Figure IV-18. In the productions, x represents any node identifier except s and t. The token "[m]" represents a reduction of a subexpression containing m paths. On the left side of the productions, the symbol "+" represents a union operation; on the right side it represents arithmetic addition. Likewise, on the left side of the productions, concatenation represents concatenation; on the right it represents multiplication. Production (1) simply converts each alphabet symbol into a unit path count. Production (2) represents the reduction of a concatenation of one subexpression containing m paths and another subexpression containing n paths. The concatenation produces mn many possible paths. Production (3) represents the reduction of a union of two subexpressions, one with m paths and the other with n paths. Since either subexpression, but not both, may be traversed, the total paths represented is m+n. Production (4) simply removes parentheses. Production (5)

(1)	x	→	[1]
(2)	[m][n]	→	[mn]
(3)	[m]+[n]	→	[m+n]
(4)	([m])	→	[m]
(5)	[m]*	→	[m+1]
(6)	s[n]t	→	n

Figure IV-18. Productions for reducing regular expression

represents the m many ways through an iteration plus the one path around it. Production (6) halts the reductions and causes the path count to be output. The algorithm in Figure IV-19 is an implementation of the productions.

```
procedure ComputeNumberPaths (RE: RegExpType; var NumPaths: integer);
 var token: alphabettokentype;

 function expression: integer;
  var x: integer;
  begin
   x := term;
   if token = '+' then
      token := next (RE, token);
      expression := x + expression
   else expression := x
  end;

 function term;
  var x: integer;
  begin
   x := factor;
   if token in REalphabet ∪ {"("} then
     term := x * term
   else term := x
  end;

 function factor;
  var x: integer;
  begin
   if token = '(' then
      token := next (RE, token);
      x := expression;
      token := next (RE, token)
   else if token in REalphabet then
      x := 1;
      token := next (RE, token)
   endif;
   if token = '*' then
      x := x + 1;
      token := next (RE, token)
   endif;
   factor := x
  end;

 begin
  token := firsttoken(RE);
  NumPaths := expression
 end;
```

Figure IV-19. Procedure to compute a path count from a regular expression

A path tree and a regular expression each provide benefits that the other does not. Regular expressions are compact and allow the use of pattern matching tools for control flow analysis. The path tree provides a two dimensional view of the structure of execution paths in a flowgraph, showing immediately the degrees of iterative nesting possible. Thus, they are complementary tools for analyzing paths in programs.

2. An observation about path counts

This section presents an interesting observation on the difference in the numbers of paths that can be generated from structures of *while* loops and structures of *repeat* loops. The number of paths, satisfying Criterion C_5, that can be generated from these structures demonstrates the impracticality of using such a path set as an aid to understanding the control structure of a program. Upper and lower bounds on the number of paths that can be computed from structures containing the two elementary constructs are derived below.

A single *while* loop represents two paths: one if the loop is traversed, and a second if it is not. If a construct containing k paths is nested as the body of a *while* loop, then the new construct represents k+1 paths, the k paths through the body of the loop plus the one if the outer loop is not traversed. A simple induction shows that a structure comprised of n *while* loops, nested such that $loop_i$ forms the body of $loop_{i-1}$, represents n+1 paths.

If those n *while* loops are placed in a sequence, such that none of them are nested, each loop doubles the number of paths in the part of the construct preceding it. Another induction shows that this construct represents 2^n paths.

If the fully nested construct is rearranged so that at least one of the loops is placed in sequence with another, then that one loop no longer adds just one more path to the total path count. Its contribution is multiplicative instead of additive. Thus,

the minimum number of paths that can be generated by a structure comprised of n *while* loops is n+1. Likewise, if the fully sequenced construct is rearranged such that some nesting is introduced, then the number of paths represented by the new construct must be less than 2^n. The contribution of the nested loop becomes additive instead of multiplicative. Therefore, the upper bound on the number of paths that can be generated from n *while* loops is represented by the fully sequenced structure.

A structure of n fully sequenced *repeat* loops also represents 2^n paths, for the same reasons as the *while* loops. However, this value represents the lower bound on the number of paths that can be generated by n *repeat* loops, instead of the upper bound. Nested *repeat* loops create far more paths. Consider the flowgraph in Figure IV-14(a). The single path that contains no iteration is s-w-x-y-t. A second path, s-w-x-x-y-t, reflects an iteration of just the inner loop. If the outer loop is iterated, then the inner loop can be iterated either before or after, both before and after, or neither before nor after the back arc is traversed. These possibilities produce the additional paths

s-w-x-y-z-w-x-y-t,
s-w-x-x-y-z-w-x-y-t,
s-w-x-y-z-w-x-x-y-t, and
s-w-x-x-y-z-w-x-x-y-t.

The total of six paths is twice as many as the ones represented by two nested *while* loops.

In general, the number of paths that can be generated from a structure containing n nested *repeat* loops, where $loop_i$ is nested immediatedly in $loop_{i-1}$ for $1 < i \leqslant n$, is given by the function

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n-1)(f(n-1)+1) & \text{if } n > 0 \end{cases}$$

This can be seen by a simple induction on the number of loops. If $k = 0$, then there is no loop, and the structure represents just one path (assuming no other constructs are allowed). Suppose the function holds for $k > 0$ nested *repeat* loops. That is, the

structure represents $f(k) = f(k-1)(f(k-1)+1)$ paths. Enclose this structure within a k+1st *repeat* loop. The number of paths in the new structure can be computed as follows. There are, by the induction hypothesis, $f(k)$ many paths through the structure if the outer loop is not traversed. If the outer loop is traversed, then there are $f(k)$ many paths from s to the loop's predicate, and $f(k)$ paths from the predicate through the loop's body and back to the predicate, for a total of $f^2(k)$ paths. The sum of all the paths is $f(k) + f^2(k) = f(k)(f(k)+1) = f(k+1)$. For $k = 5$, $f(k) =$ 3,263,442, five orders of magnitude more than the maximum of 32 paths that can be represented by five *while* constructs. Providing a programmer an enumeration of that many paths will do little to help clarify the control structure of a program.

V. ALGORITHMS AND EXAMPLES

This chapter presents the algorithms that compute the four properties and their corresponding measures defined in Chapters III and IV. Following the algorithms is an assessment of their computational complexity. For the algorithms to be a useful programming environment tool, they must run in a reasonable amount of time. The analysis shows that they do. This is encouraging because they are straightforward implementations of the definitions. Following the run time analyses, Section C presents examples of applications of the properties and their measures to demonstrate their utility.

A. Algorithms

The algorithms are presented using a high-level pseudo-language and abstract data types. Input to the algorithms is a representation of the flowgraph of a program. The node is assumed to be a basic data type on which other data structures are based. The abstract data types used for the first set of algorithms are defined in Figure V-1. Associated with these types are invariants that stem immediately from the definition of flowgraph. They include properties such as each "from" and "to" in the arc set

```
NodeSet = set of NodeType;
ArcType = record
      From, To: NodeType
    End;
ArcSet = set of ArcType;
Flowgraph = record
          Nodes: NodeSet;
          Predicates: NodeSet;
          Arcs: ArcSet;
          Start, Exit: NodeType
        end;
NodeSetList = array [NodeType] of NodeSet;
NodeList    = array [NodeType] of NodeType;
ListofIntegers = array [NodeType] of integer;
ListofReals = array [NodeType] of real;
```

Figure V-1. Abstract data types

must appear in the node set, and for each node x in the node set there is a sequence of arcs in the arc set that defines a path from the start node to x. Additionally, for each node x $\in$ Predicates there are at least two distinct arcs y and z in Arcs where y.from = z.from = x. Operations on the abstract type Flowgraph include the functions Predecessors(x) which returns the set of immediate predecessors of node x and Successors(x) which returns the set of immediate successors of node x.

The first step in computing the PEN, NLP and UNST measures is the computation of the ranges of the predicate nodes in flowgraph G. This process has two steps: determination of the greatest lower bounds (GLBs), and computation of the Ranges.

The determination of GLB(p) first requires the identification of all of the lower bounds of p. That the lower bounds of p are the inverse dominators of p suggests a method for generating lower bounds. The arcs in G can be reversed creating GInverse, the dominators for each node in GInverse can be computed, and then those dominators can be assigned as the lower bounds of those nodes in G.

Several efficient dominator algorithms appear in the open literature. The one presented is due to Aho and Ullman [43]. Lengauer and Tarjan [44] have developed a more generally efficient algorithm, but because computing dominators takes only a small part of the processing required to compute the measures, the one due to Aho and Ullman suffices. Their algorithm is given as Algorithm A.

Algorithm B, GenGLBs, inverts flowgraph G and calls FindDominators to obtain the sets of lower bounds. GLB(p) for each predicate p is p's immediate dominator in GInverse. To find that immediate dominator, GenGLBs performs a depth-first search from each predicate node to node t. The first node in the predicate's lower bounds set encountered in the search is the greatest lower bound. Since it lies on all paths from the predicate node to t, it must be encountered. The search is terminated when the GLB is found.

```
Procedure FindDominators (G: Flowgraph; var Dom: NodeSetList);
  var  NewDom: NodeSet;
       M, N: NodeType;
       Changed: boolean;

(* pre: True  *)
(* post: ∀ i [ i ∈ G.Nodes ⇒  x ∈ Dom[i] iff x dominates i in G] *)

begin
  Dom[G.Start] := {G.Start};
  for each node N ∈ G.Nodes-{G.Start} do
    Dom[N] := G.Nodes;
  repeat
    Changed := false;
    for each N ∈ G.Nodes - {G.Start} do
      NewDom := G.Nodes;
      for each M ∈ Predecessors(N) do
        NewDom := (NewDom ∩ Dom[M]) ∪ {N};
      if NewDom ≠ Dom[N] then
        Changed := true;
        Dom[N] := NewDom
      endif
    endfor
  until not Changed
end;
```

Figure V-2. Algorithm A

```
Procedure GenGLBs (G:Flowgraph; var GLBs: NodeSetList);
var GInverse: Flowgraph;
  A: ArcType;
  N, M: NodeType;
  LowerBounds: NodeSetList;
(* pre: True  *)
(* post: ∀i [ i ∈ G.Nodes ⇒  GLBs[i] = GLB(i) in G *)
begin
  GInverse.Nodes := G.Nodes;
  GInverse.Arcs := Ø;
  for each A in G.Arcs do
    Ginverse.Arcs := GInverse.Arcs ∪ {(A.To, A.From)}
  GInverse.Start := G.Exit;
  GInverse.Exit := G.Start;
  FindDominators (GInverse, LowerBounds);
  for each node N in G.Nodes do
    S1 := {N};
    GLBs[N] := G.Start;
    repeat
      S2 := S1;  S1 := Ø;
      for each node M in S2 do
        if Successors(M) ∩ LowerBounds[N] = Ø then
          S1 := S1 ∪ Successors(M)
        else
          GLBs[N] := Successors(M) ∩ LowerBounds[N];
      endfor
    until GLBs[N] ≠ G.Start
  endfor
end;
```

Figure V-3. Algorithm B

To generate the Range of each node n, FindRanges, Algorithm C, uses a depth-first search to traverse all of the paths from the immediate successors of n to GLB(n). It places each node encountered on the traversal into Range(n).

FindRanges can also generate the predicators of each node. As it encounters each node x in Range(n), it can add n to Pred(x). This is implemented by inserting the statements

```
for  each node J in G.Nodes do
  Pred[J] := Ø;
```

immediately after the "begin" for FindRanges, and adding

```
Pred[X] := Pred[X] ∪ {N};
```

immediately after the statement commented as #1 in RangeDFS. The declaration "var

```
Procedure FindRanges (G: Flowgraph;  Glb:NodeList;
                      var Range, Range': NodeSetList);
var I, J: NodeType;
(* pre: ∀i [ i ∈ G.Nodes ⇒  Glb[i] = GLB(i)] in G *)
(* post: ∀i [ i ∈ G.Predicates ⇒  Range[i] = Range(i)
                        ∧ Range'[i] = Range'(i) in G] *)

  Procedure RangeDFS (N, X: NodeType);
  var M: NodeType;
(* pre: X ≠ Glb[N] ∧ X ∉  Range[N] *)
(* post: X and all nodes on paths between X and Glb[N] are in Range[N] *)
  begin
     Range[N] := Range[N] ∪ {X};  (*  #1  *)
     for each M in Successors(X) - {Glb[N]} do
       if M ∉  Range[N] then
         RangeDFS (N, M)
  end;

begin (* FindRanges *)
  for each I in G.Predicates do
    Range[I] := ∅;
    for each J in Successors(I) - {Glb[I]} do
      if J ∉  Range[I] then
        RangeDFS (I, J)
    endfor;
    Range'[I] := Range[I] ∪ {I}
  endfor
end;
```

Figure V-4. Algorithm C

Pred: NodeSetList;" must also be added to the parameter list for FindRanges.

The computation of PEN can also be implemented in Algorithm C. For each node x that RangeDFS adds to Range(n), it can add 1 to PEN(x). To implement this, the statements

```
for each node J in G.Nodes do
  PEN[J] := 0;
```

are inserted immediately after the "begin" for FindRanges, and the statement

```
PEN[X] := PEN[X] + 1;
```

is inserted immediately after the statement commented as #1 in RangeDFS. The declaration "var PEN: ListofIntegers;" must also be added to the parameter list of

FindRanges.

Computing NLP is more involved. After determining that node n falls in the range of a loop predicate p, Algorithm D must ensure that n lies on a cycle delimited by p. The function ThereIsaPath checks for this by performing a breadth-first search of Range(p) beginning at node n. If node p is encountered then there must be a path from n to p, and the function returns true. Otherwise, it returns false.

To compute UNST(x) for each node x, the relative unstructuredness of each predicate pair in Pred(x) is needed. Since the structuredness of a predicate pair depends on the entry point for each range, the entry points are generated first. The FindEntry procedure, Algorithm E, is a straightforward implementation of the entry node definition.

The function OptEntry, Algorithm F, chooses an optimal entry node for the multiple-entry loop in Range'(n). It considers all nodes in Range'(n) that have inarcs originating outside the Range'. For each such candidate node x, OptEntry determines which of the predicates q in Pred(n) are such that all paths from q into Range'(n) enter Range'(n) at x. OptEntry returns the candidate with the most predicates satisfying this "all-paths" condition.

The function AllPaths, Algorithm G, returns true if all the paths from predicate k ($k \in Pred(i)$) into Range'(i) enter the range' at node j. The algorithm computes the set of nodes in Range'(k) reachable from node k when all inarcs to node j are removed. If the set of nodes reachable from node k contains any nodes in Range'(i) then there must be a path from k into Range'(i) that does not contain node j, and AllPaths returns false. Otherwise, all paths from k into Range'(i) contain j and the function returns true.

The procedure to determine the structuredness of a predicate pair, Algorithm H, is a straightforward implementation of the structuredness definition.

```
Procedure ComputeNLP (G: Flowgraph; Ranges, Pred: NodeSetList;
                      var LPred: NodeSetList;  var NLP: ListofIntegers):
var M, N: NodeType;
(* pre: ∀i [ i ∈ G.Nodes ⇒  Ranges[i] = Range(i) in G
        ∧ Pred[i] = Pred(i) in G]  *)
(* post: ∀i [ i ∈ G.Nodes ⇒
        LPred[i] = {p | ∃L [L is a cycle delimited by p ∧ i ∈ L]}
        ∧ NLP[i] = |LPred[i]| ]  *)

  Function ThereIsaPath (N, LoopPredicate: NodeType;
                         R: NodeSet): boolean;
  var S, OldS, NewS: NodeSet;
    M: NodeType;
  (* pre: R = Range(LoopPredicate) ∧ N ∈ Range(LoopPredicate)  *)
  (* post: ThereIsaPath = true iff LoopPredicate was encountered in
           a breadth-first search of Range(LoopPredicate)
           beginning at node N  *)
  begin
    S, NewS := {N};
    repeat
      OldS := NewS;
      for each M in S do
        NewS := NewS ∪ (Successors(M) ∩ R);
      S := NewS - OldS
    until (LoopPredicate ∈ NewS) or (S = ∅);
    ThereIsaPath := LoopPredicate ∈ NewS
  End;  (* of ThereIsaPath *)

begin (* ComputeNLP *)
  for each N in G.Nodes do
    LPred[N] := ∅;
    NLP[N] := 0;
    for each M in Pred[N] do
      if (M ∈ Ranges[M]) and ThereIsaPath (N, M, Ranges[M]) then
        LPred[N] := LPred[N] ∪ {M};
        NLP[N] := NLP[N] + 1
      endif
  endfor
end;
```

Figure V-5. Algorithm D

```
Procedure FindEntry (G: Flowgraph;  Range', Range, Pred: NodeSetList;
                     var Entry: NodeList);
var I: NodeType;
  D: NodeSet;
  Dom: NodeSetList;
(* pre: ∀x [ x ∈ G.Nodes ⇒  Pred[x] = Pred(x) in G
                ∧ Range[x] = Range(x) in G]
        ∧ ∀y [y ∈ G.Predicates ⇒  Range'[y] = Range'(y) in G]  *)
(* post: ∀ x [ x = Entry[y] ⇒  node x satisfies the definition
      of "entry node" for Range'(y)]  *)
begin
  FindDominators (G, Dom);
  for each I in G.Predicates do
    D :=      ⋂      Dom[n];
          n∈Range'[I]
    if D  ∩ Range'[I] ≠ ∅ then
    (*the intersection is either empty or a singleton set *)
      Entry[I] := x where {x} = D ∩ Range'[I]
    else
      if I ∈ Range[I] then
        (* Range'[I] contains a multiple entry loop *)
        Entry[I] := OptEntry (G, Range', Pred[I], I)
      else
        (* node I is an alternative predicate  *)
        Entry[I] := I
      endif
    endif
  endfor
end  (* FindEntry  *)
```

Figure V-6. Algorithm E

```
Function OptEntry (G: Flowgraph; Range': NodeSetList;
                   Pred: NodeSet;
                   PredNode: NodeType): NodeType;
var Maxcount, Count: integer;
  J, K: NodeType;
(* pre: Range'(PredNode) contains a multiple-entry loop predicated by PredNode
      ∧ ∀i [i ∈ G.Nodes ⇒ Range'[i] = Range'(i) in G]
      ∧ Pred = Pred(PredNode) in G   *)
(* post: OptEntry = node n such that all paths from the most
       predicates in Pred(PredNode) enter Range(PredNode) through n *)
begin
  Maxcount := 0;
  for each J in Range'[PredNode] do
    if Cardinality (Predecessors(J)-Range'[PredNode]) > 0 then
      (* J is a candidate entry node  *)
      Count := 0;
      for each K in Pred do
        if Range'(K) ≠ Range'[PredNode] then
          if AllPaths (G, Range'[K], Range'[PredNode], K, J) then
            Count := Count + 1;
      if Count > Maxcount then
        Maxcount := Count;
        OptEntry := J
      endif
    endif
  endfor
end; (* of OptEntry *)
```

Figure V-7. Algorithm F

```
Function AllPaths (G: Flowgraph; Range'K, Range'I: NodeSet;
                   K, J: NodeType): boolean;
var NewReachables, OldReachables: NodeSet;
  N: NodeType;
(* pre: Range'I ⊂ Range'K ∧ J ∈ Range'I ∧ K ∈ Range'K *)
(* post: AllPaths = true iff J falls on all paths from K into Range'I *)
begin
  NewReachables := Successors(K) - {J};
  repeat
    OldReachables := NewReachables;
    for each N in OldReachables do
      NewReachables := NewReachables ∪ (Successors(N) ∩ Range'K) - {J}
  until OldReachables = NewReachables;
  AllPaths := NewReachables ∩ Range'I = ∅
end  (* of AllPaths *)
```

Figure V-8. Algorithm G

```
Procedure Structuredness (G: Flowgraph;  P, Q: NodeType;
                          Pred, Range': NodeSetList;
                          Entry: NodeList;
                          var Structured: NodeSetList);
var X: NodeType;
(* pre: {P,Q} ⊆ G.Predicates ∧ P ≠ Q
      ∧∀i [i ∈ G.Nodes ⇒  Range'[i] = Range'(i) in G]
      ∧∀i [i ∈ G.Predicates ⇒  Pred[i] = Pred(i) in G]
      ∧∀i [i ∈ G.Predicates ⇒  Entry[i] = Entry(i) in G ] *)
(* post: P ∈ Structured[Q] ∧ Q ∈ Structured[P]
       iff P is structured with respect to Q *)

  Procedure Struct (I, J: NodeType;  var Structured: NodeSetList);
    var N: NodeType;
    begin
(* pre: True  *)
(* post: I ∈ Structured[J] ∧ J ∈ Structured[I]  *)
      Structured[I] := Structured[I] ∪ {J};
      Structured[J] := Structured[J] ∪ {I}
    end;

begin
  if Range'[P] ∩ Range'[Q] = ∅ then
    Struct (P, Q, Structured)
  else if (Range'[P] ≠ Range'[Q]) and
        (P ∈ Range'[Q] or Q ∈ Range'[P]) then
    begin
      if Q ∈ Range'[P] then
        Swap (P, Q);     (*  P ∈ Range'(Q)  *)
      if AllPaths (G, Range'[Q], Range'[P], Q, Entry[P]) then
        Struct (P, Q, Structured)
      else
          for each X in Range'[Q] ∩ G.Predicates do
            if Range'(Q) ≠ Range'(X) and
               AllPaths (G, Range'[Q], Range'[P], Q, Entry[X]) then
                Struct (P, Q, Structured);
    end
end;  (* Structuredness *)
```

Figure V-9. Algorithm H

The computation of UNST needs the number of structured predicate pairs in each Pred set. This can be computed by adding the statements

```
for each X in G.Nodes do
  if {I, J} ⊆ Pred[X] then
    NumStructPairs[X] := NumStructPairs[X] + 1;
```

to the procedure Struct, and adding the declaration "var NumStructPairs: ListofIntegers" to the parameter list for Structuredness. UNST(x) can be computed

for each node x in G as shown in Algorithm I.

The last measure, NAnte, is computed by Algorithm J. For each node in flowgraph G, the procedure computes the closure set of its predecessors. The algorithm is very similar to Aho and Ullman's dominator algorithm, Algorithm A. It initializes each node's predecessor set to its set of immediate predecessors. It propagates the predecessors by taking the union of the predecessor sets belonging to each node's immediate predecessors. It repeats this until no new predecessors are added to any set.

B. Computational Complexity

This section argues the computational complexities of the algorithms presented in Section A. Since sets are widely used in these algorithms, if two simultaneous implementations, such as bit vector and linked list, would significantly speed up the set operations, both are assumed to be used. In the following analyses, N, P, and E represent the number of nodes, number of predicate nodes, and number of edges,

```
Procedure ComputeUNST (G: Flowgraph; Pred: NodeSetList;
                       NumStructPairs: ListofIntegers;
                       var Unst: ListofReals):
var Totalpairs: integer;
  X: NodeType;
(* pre: ∀i [i ∈ G.Nodes ⇒  Pred[i] = Pred(i) in G
       ∧ NumStructPairs[i] = number of structured predicate pairs
          in Pred[i]] *)
(* post: ∀i [i ∈ G.Nodes ⇒  Unst[i] = degree of structuredness
                    of Pred[i]  *)
begin
  for each node X in G.Nodes do
    Totalpairs := Cardinality(Pred[X]) * (Cardinality(Pred[X]) -1) / 2;
    if Totalpairs > 0 then
      Unst[X] := 1 - NumStructPairs[X] / Totalpairs
    else
      Unst[X] := 0
  endfor
end;
```

Figure V-10. Algorithm I

```
Procedure ComputeAnte (G: Flowgraph;  var Ante: NodeSetList;
                       var NAnte: ListofInteger;
  var  N, M: NodeType;
       Changed: boolean;
       S: NodeSet;

(* pre:  true *)
(* post: ∀i [i ∈ G.Nodes ⇒  ∀x [x ∈ Ante[i] iff x precedes i in G]
       ∧ NAnte[i] = | Ante[i] | ]  *)

begin
  for each node N in G.Nodes do
     Ante[N] := Predecessors(N);
  repeat
     Changed := false;
     for each node N in G.Nodes do
        S := Ante[N];
        for each node M in Predecessors(N) do
           S := S ∪ Ante[M]
        endfor;
        if S ≠ Ante[N] then
           Changed := true;
           Ante[N] := S
        endif;
     endfor
  until not Changed;
  for each node N in G.Nodes do
     NAnte[N] := Cardinality (Ante[N])
end;
```

Figure V-11. Algorithm J

respectively, in flowgraph G. Because the expected running time of some of the algorithms is significantly greater for irreducible flowgraphs (flowgraphs that contain multiple entry loops), than for reducible flowgraphs, complexities for processing each will be given.

In algorithm A, each pass through the *repeat* loop processes each node and arc once, resulting in an O(N+E) run time. When processing irreducible flowgraphs, the repeat may have to iterate N times to propagate all the dominator information to all the nodes. Thus, for irreducible flowgraphs, FindDominators takes O(N(N+E)) time. Reducible flowgraphs require only two passes through the *repeat* loop if the nodes are chosen by the outer *for* statement in reverse postorder number [13] sequence. Reverse postorder numbers are assigned during a depth first search of the flowgraph. Since the

search requires only O(N+E) time, FindDominators can process reducible flowgraphs in O(N+E) time. Lengauer and Tarjan's [44] dominator algorithm runs in O(Eα(E,N)) time for all flowgraphs, where α(E,N) is a functional inverse of Ackermann's function [45]. In the rest of the analyses, this algorithm is the one assumed to be used to find dominators in irreducible flowgraphs.

Algorithm B, GenGLBs, first requires O(E) time to create GInverse. Then, after the dominators of GInverse have been found, the greatest lower bounds are computed by following a path from each node to its GLB. This requires O(N+E) time for one node, and O(N(N+E)) time for all N nodes. Thus, the total time needed by GenGLBs, including the time needed to compute the dominators, is O(E+(N+E)+N(N+E)) for flowgraphs without multiple-exit loops, and O(E+Eα(E,N)+N(N+E)) for flowgraphs with multiple-exit loops. Multiple-exit loops cause the inverse of a flowgraph to be irreducible.

To generate ranges, FindRanges (Algorithm C) performs an O(N+E) depth-first search for each predicate node, resulting in an O(P(N+E)) time requirement. Adding the computations for Pred and PEN to FindRanges adds an O(N) array initialization, increasing the running time to O(P(N+E)+N). The reducibility of a flowgraph does not affect the running time of this algorithm.

The procedure ComputeNLP, Algorithm D, checks, for each of the N nodes in the flowgraph, if there is a path from that node to each of its possibly P loop predicates. The path check function performs an O(N+E) breadth-first search. To compute the NLP measure for each node requires O(PN(N+E)) time.

The procedure FindEntry, Algorithm E, begins by computing the dominators in G. Then, for each of the P ranges, it computes the immediate dominator of the set of nodes in the range. This can require N dominator set intersections per predicate node. If the flowgraph is reducible, no further processing of the flowgraph is needed. So, for reducible flowgraphs, the algorithm takes O(N+E+PN) time. If the flowgraph is

irreducible, then for possibly all of the P predicates, OptEntry must be called. For each of the possibly N nodes in the range being processed by OptEntry, the algorithm checks if all the paths from every predicator of the range enters the range at that node. This "all"paths check is performed by the function AllPaths (Algorithm G), which performs an O(N+E) breadth-first search each time called. As a result, OptEntry requires up to O(NP(N+E)) time each time it is called. Since it could be called once for each of the P predicates, FindEntry has a worst case run time of $O(P^2N(N+E))$ to compute all the optimal entry points. Adding the time needed to find the dominators, and the dominators of each range, increases the time complexity to $O(E\alpha(E,N)+PN+P^2N(N+E))$ for irreducible flowgraphs.

To determine the structuredness of predicates p and q, the procedure Structuredness may first have to check if all paths from p into Range'(q) contain Entry(q), an O(N+E) breadth-first search. Then, it may have to check if all the paths from p go through the entry nodes of the possibly P many ranges that contain Range'(q). Again, an O(N+E) breadth-first search is used, resulting in a O(P(N+E)) time requirement. For both steps, the time requirement is O((N+E)+P(N+E)), which holds for reducible and irreducible flowgraphs alike. Since there are ½P(P-1) many predicate pairs in a flowgraph, the time needed to determine the structuredness of all the pairs is O(½P(P-1)((N+E)+P(N+E))). Adding the code to save the number of structured pairs for each node, adds an O(N) loop which increases the time required to O(½P(P-1)((N+E)+P(N+E)+N)).

To compute UNST, Algorithm I performs O(1) mathematical operations for all of the nodes in G. This results in an O(N) expected running time.

Because the procedure ComputeAnte, Algorithm J, uses the same basic algorithm as Aho and Ullman's dominator algorithm, the time complexity is the same. However, selecting nodes in reverse post order sequence does not improve the time complexity for computing predecessors. Therefore, ComputeAnte requires O(N(N+E))

for all flowgraphs.

For reducible flowgraphs, determining the structuredness of all the predicate pairs in the flowgraph is the dominant computation. For irreducible flowgraphs, determining entry points and computing the relative structuredness of the predicate pairs are the dominant computations. The upper bound for processing either type of flowgraph is $O(N^3E)$. Table V-1 summarizes the complexities of the algorithms.

C. Examples

In this section several flowgraphs are presented to illustrate applications of the structuredness definition, and several others for which the measures are computed. Shown first is that the structuredness definition accurately reflects McCabe's [2] and Williams' [28] assessment of the four basic unstructured constructs. The constructs are shown in Figure II-10. The range of each predicate in those flowgraphs is listed in Table V-2. In the discussion of each flowgraph, the conditions referred to are the ones numbered in the structuredness definition, Definition 3.13.

In flowgraph (a), Range'(b) is a proper subset of Range'(a); therefore, conditions (1) and (2) of the definition do not apply. Condition (3) is not satisfied because two arcs from node a enter Range'(b) at two distinct nodes, b and c. There is no third

Table V-1. Summary of complexities of the algorithms

Algorithm	Reducible Flowgraph	Irreducible Flowgraph
Dominators	$O(N+E)$	$O(E\alpha(E,N))$
GLBs	$O(E+(N+E)+N(N+E))$	$O(E+E\alpha(E,N)+N(N+E))$
Ranges	$O(N+P(N+E))$	$O(N+P(N+E))$
NLP	$O(NP(N+E))$	$O(NP(N+E))$
AllPaths	$O(N+E)$	$O(N+E)$
OptEntry	not called	$O(NP(N+E))$
FindEntry	$O(N+E+PN)$	$O(E\alpha(E,N)+PN+P^2N(N+E))$
Structured		
One pair	$O((N+E)+P(N+E)+N)$	$O((N+E)+P(N+E)+N)$
All pairs	$O(\frac{1}{2}P(P-1)((N+E)+P(N+E)+N))$	$O(\frac{1}{2}P(P-1)((N+E)+P(N+E)+N))$
UNST	$O(N)$	$O(N)$
Ante	$O(N(N+E))$	$O(N(N+E))$

Table V-2. Range information for the flowgraphs of Figure II-10

Flowgraph	Predicate	Range'
(a)	a	{a,b,c}
	b	{b,c}
(b)	a	{a,b}
	b	{a,b}
(c)	a	{a,b}
	c	{b,c}
(d)	b	{a,b}
	c	{a,b,c}

predicate to satisfy condition (4); therefore, predicate a is unstructured with respect to predicate b.

Flowgraph (b) illustrates the multiple-exit loop. Both a and b branch out of the cycle a-b-a. As a result, Range'(a) = Range'(b), satisfying condition (2). Thus, predicate a is unstructured with respect to predicate b.

In flowgraph (c), condition (1) does not apply because Range'(a) and Range'(c) are not disjoint; they share node b. Neither predicate falls within the Range' of the other, satisfying condition (2); therefore, predicate a is unstructured with respect to predicate c. Note that this construct also violates condition (3), because arcs from node a enter Range'(c) at two distinct nodes, b and c.

In flowgraph (d), Range'(b) is entirely nested within Range'(c); therefore, conditions (1) and (2) do not apply. Again, there is no third predicate, so condition (4) does not apply. However, condition (3) does apply, and is violated. The arc from c to b is a path into Range'(b) that does not contain Entry(b), node a. Thus, predicate b is unstructured with respect to predicate c.

The structuredness definition, then, accurately reflects the unstructuredness contained in each of the four constructs.

The second example, the flowgraph shown in Figure V-12, contains two constructs, Range'(p3) and Range'(p4), for which entry nodes must be chosen. Range'(p3) = {p3,w,z} and Range'(p4) = {p3,p4,w,z}. The candidate entry nodes for

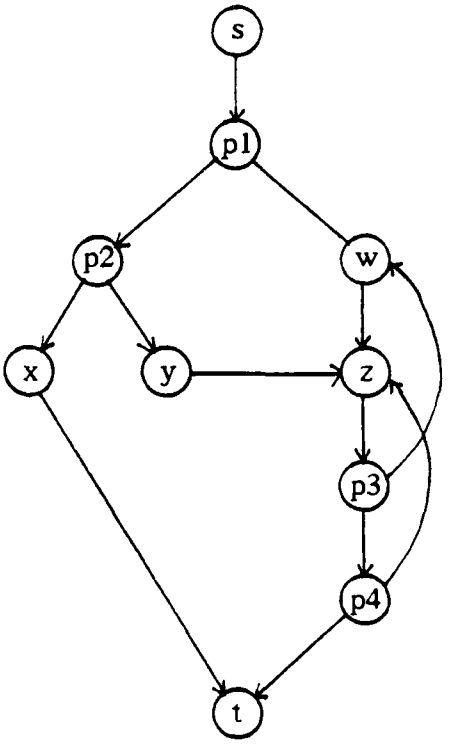


Figure V-12. An unstructured flowgraph

both are w and z. Consider Range'(p3) first. No predicate has all its paths entering the range at node w. However, all paths from p2 and p4 enter at node z. Therefore, node z is chosen as Entry(p3). Now consider Range'(p4). Again, no predicate has all its paths entering the range at w. But all paths from p2 enter at z. Thus, node z is again chosen as Entry(p4). This choice of entry nodes tells the programmer to view the graph in Figure V-12 as if it were drawn as in Figure V-13. That is, the flowgraph should be viewed as if there is an unstructured *goto* from p1 into the ranges of p2, p3 and p4.

The choice of optimal entry node does not always provide such an alternative view of a flowgraph. Consider the flowgraph in Figure V-14 which contains the multiple-entry Range'(p4), where Range'(p4) = {x,p4}. Again, the optimal entry

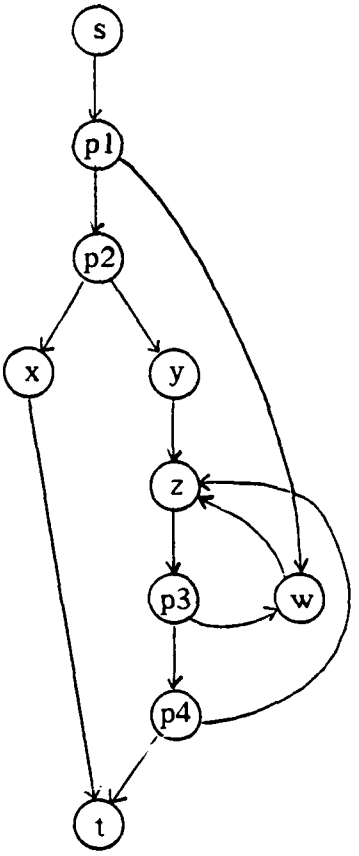


Figure V-13. An "optimal" view of the flowgraph in Figure V-12

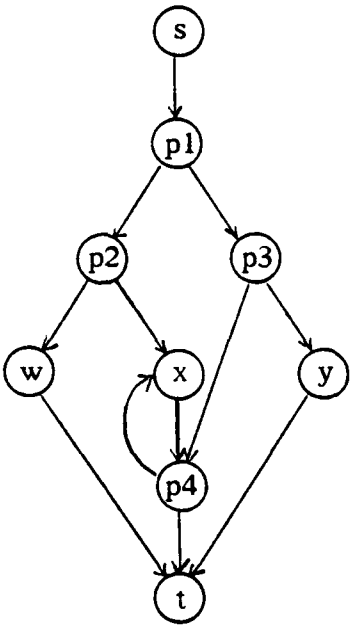


Figure V-14. An unstructured flowgraph

node for Range'(p4) needs to be determined. The candidate nodes are x and p4. Node x is contained on all paths from only predicate p2 into Range'(p4). Node p4 is contained on all paths from only predicate p3 into Range'(p4). Each candidate node has only one corresponding "all paths" predicate; therefore, there is no optimal choice. The choice of entry node is arbitrary; the degree of unstructuredness of p4's control flow context will be the same regardless of the choice of Entry(p4). The flowgraph can be viewed as having Range'(p4) nested in Range'(p3) with an explicit branch from p2 to x, or as having Range'(p4) nested in Range'(p2) with an explicit branch from p3 to p4. Although the structuredness property cannot always provide a "better" view of a flowgraph, it does provide the least unstructured view when possible.

Now applications of the vector of measures are illustrated. Considered first are three flowgraphs for a table search algorithm, two from Knuth [10] and the third from Gileadi and Ledgard [46]. The flowgraphs are shown in Figure V-15, and their

respective measures are listed in Table V-3. The two predicates in flowgraph (a) are unstructured, but all predicate pairs in (b) and (c) are structured. The measures show that flowgraph (b) has the least involved control flow because it is structured, has the least nesting and has the fewest predecessors per node. Flowgraph (c), however, is an equally likely solution and illustrates the trade-offs that often arise with structuring. Although it is structured and contains just one loop, flowgraph (c) contains much more nesting and many more paths than the unstructured flowgraph (a). If one had to choose between the implementations represented by flowgraphs (a)

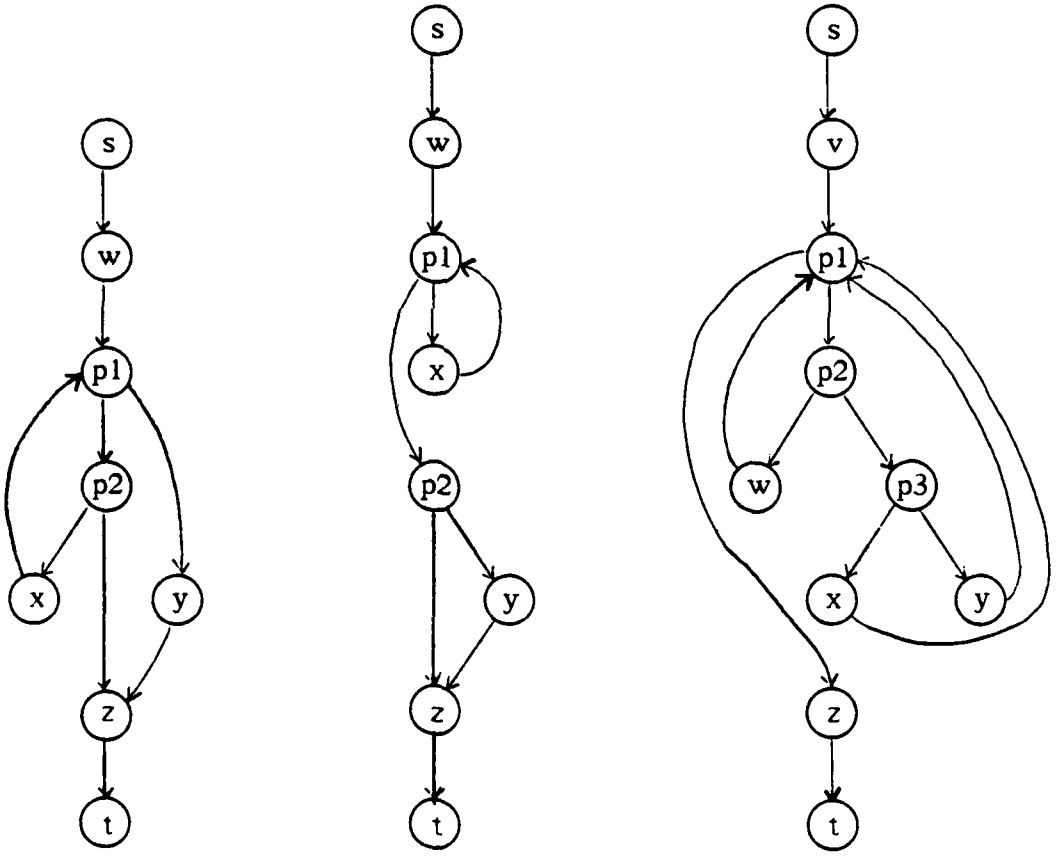


Figure V-15. Flowgraphs for three table search algorithms

Table V-3. Measures for the flowgraphs of Figure V-15

Flowgraph	Node	Pred	PEN	NLP	UNST	NAnte
(a)	w	Ø	0	0	0	1
	p1	{p1,p2}	2	2	1	5
	p2	{p1,p2}	2	2	1	5
	x	{p1,p2}	2	2	1	5
	y	{p1,p2}	2	0	1	5
	z	Ø	0	0	0	6
(b)	w	Ø	0	0	0	1
	p1	{p1}	1	1	0	4
	x	{p1}	1	1	0	4
	p2	Ø	0	0	0	4
	y	{p2}	1	0	0	5
	z	Ø	0	0	0	6
(c)	v	Ø	0	0	0	1
	p1	{p1}	1	1	0	8
	p2	{p1}	1	1	0	8
	w	{p1,p2}	2	1	0	8
	p3	{p1,p2}	2	1	0	8
	x	{p1,p2,p3}	3	1	0	8
	y	{p1,p2,p3}	3	1	0	8
	z	Ø	0	0	0	8

and (c), the choice would be between unstructuredness and nesting. Flowgraph (c) illustrates that even well-structured algorithms can contain constructs that have significantly more involved control flow than functionally equivalent unstructured versions.

The next four flowgraphs illustrate how slight changes in control flow affect the values in the vector of measures. The flowgraph in Figure V-16(a) is a simple nested *if-then-else* construct. The vector values for nodes p2 and b are:

	PEN	NLP	UNST	NAnte
p2	1	0	0	2
b	2	0	0	3

The measures reflect that node b is predicated by 2 nodes and p2 is predicated by just one. Neither is contained in a loop, their control contexts are structured, and there is one more predecessor of b than p2.

In Figure V-16(b), p2 is changed from an *if-then-else* predicate into a *while* predicate with node b as the body of the loop. The measures for nodes p2 and b in

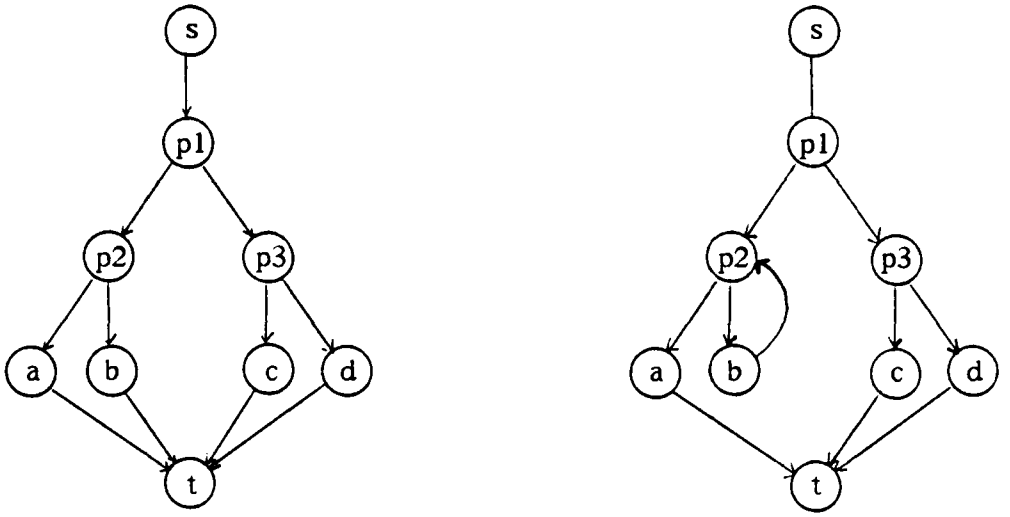


Figure V-16. Example flowgraphs

this graph are:

	PEN	NLP	UNST	NAnte
p2	2	1	0	4
b	2	1	0	4

Node b is still predicated by 2 nodes, but here one of them is a loop predicate. Node p2 is now also predicated by two nodes, itself and p1, and falls within a loop. The control flow is still structured, but the number of predecessors of p2 has doubled. Because of the loop, p2 and b are now also predecessors of p2. Node b has just one additional predecessor, itself, because of the loop.

The flowgraph in Figure V-17(a) is created from Figure V-16(a) by deleting the arc (c, t) and adding the arc (c, b). The resulting measures are:

	PEN	NLP	UNST	NAnte
p2	1	0	0	2
b	3	0	3/3	5

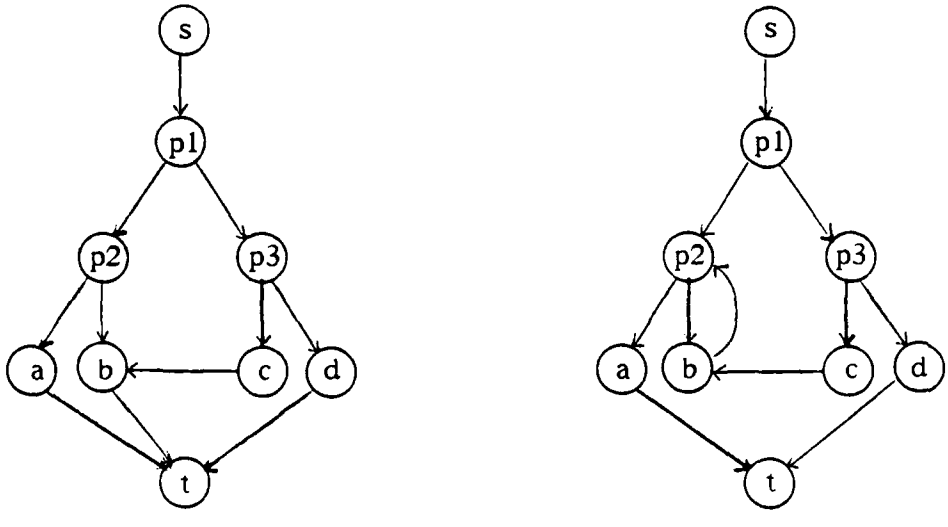
Node b is now predicated by three nodes, p1 and p2 as before, and now p3 because of the branch from c to b. p1 is unstructured with respect to both p2 and p3 because it

emits multiple paths into the range of each. The UNST value for p2 is zero because it has only one predicator; at least two are required to cause unstructuredness. Nodes p2 and p3 are unstructured because their ranges are improperly nested. The number of predecessors of node b has almost doubled because of the change in destination of node c's outarc.

The flowgraph of Figure V-17(b) is constructed from the previous one by placing nodes b and p2 into a loop predicated by p2. The vector values are now:

	PEN	NLP	UNST	NAnte
p2	3	1	2/3	6
b	3	1	2/3	6

The measures reflect both the loop containing p2 and b as well as the unstructuredness caused by the multiple paths from p1 into the loop. Nodes p2 and p3 are structured with respect to each other because OptEntry chooses node b as the entry to the loop. This accounts for the smaller UNST values. However, the number of predecessors of p2 has tripled from the previous graph. Now nodes p3, c and b can



(a) (b)

Figure V-17. Modified versions of flowgraph V-16(a)

precede p2 on some paths. The only addition to the predecessor set of node b was b itself.

From the results presented above, it would be easy to argue that the vector of measures also seems to measure complexity. The flowgraphs are presented in an order such that each one appears to be intuitively more complex than the preceding one. The measures reflect that trend. But, to argue that the measures can be used as complexity measures would nullify all the arguments previously made for the need for rigorous empirical evaluation. Although the measures seem to reflect intuitive complexity rankings, all that can really be said about them is that accurately reflect the structure of the flowgraph.

VI. CONCLUSIONS

The work reported in this dissertation is a contribution to research in software measures. It combines a more careful method with a different goal to provide an improved strategy for defining static program measures. Most earlier measures are based entirely on intuitive definitions of the properties they measure. Consequently, careful analysis and empirical validation of those measures is impossible. The properties and corresponding measures presented in this paper have been rigorously defined. This rigor facilitates analytical evaluations of the measures to ensure that they accurately reflect the property being measured.

This research does not define measures of psychological program complexity. Instead, it defines properties of program control flow that help characterize the control structure of arbitrary imperative programs. The goal is to provide a tool that helps make program control structure more discernible to a programmer. To satisfy this goal, control flow properties were defined from the perspective of a programmer, who views a program as a set of individual, interrelated components, rather than from the point of view of "complexity metrics" researchers, who tend to look at a program as a single entity to be measured. Because the most useful model of program control structure is the flow graph, the properties are defined to characterize the control flow surrounding each node in a program's flow graph. Specifically, for each node x in a flow graph, the properties identify (1) which nodes predicate x, (2) which of the predicators also predicate loops containing x, (3) which pairs of x's predicators interact in such a way that they form unstructured constructs, and (4) all the nodes that can precede x along any path from the start node to x. This information provides a clear picture of the control flow containing node x. The set of measures defined to quantify the properties can pinpoint intricate control flow and identify the properties that make it so complex.

Because the properties provide useful information and are easily computed, they can be beneficial components of automated programming environments. Automatic computation of the properties during maintenance tasks will help provide a more complete picture of the control structure of the program to be modified. The properties and their measures can also be used to identify areas of complex control flow in new programs, thereby signaling programmers to be aware of possible problem areas.

The properties and their measures are well-defined; thus, they provide analytically sound candidates for empirical evaluation as complexity measures and as tools for predicting error occurrences and programmer productivity. Unlike the situation of "complexity metrics", if empirical evaluations fail to show the measures' utility for such purposes, the properties on which the measures are based are still useful as a tool for characterizing program control structure.

The results of this research suggest several topics for future consideration. The structuredness property could be reexamined. Adaptations of the structuredness definition to allow "structured" multiple-exit loops could be made to accommodate advocates of such constructs. The algorithms could be analyzed with the intent of improving their efficiency. More efficient methods for computing the properties would enhance their utility. The most interesting area for future research is defining new criteria for finite sets of execution paths. Deeper examination of cycles, loops, and their nesting should reveal properties leading to a new definition of a path subset criterion. The new criterion would produce a set of paths containing all the information now reflected by path sets satisfying Criterion C_5. This would allow the path set and its measure to be included in the set of properties characterizing control flow. An analysis of the upper and lower bounds of numbers of paths in flow graphs constructed from other elementary constructs would be of value. Of primary interest is determining the effect of substituting multiple-exit loops for structured

ones. Computing bounds for unstructured constructs could be approached by considering each class in the hierarchy of flow graphs defined by Kosaraju [47] and Ledgard and Marcotty [48].

The methods used in this research need not be confined to measures of control flow. They can be applied to characterize other aspects of program structure. It seems almost natural that data dependency properties be identified at the node level. Properties such as the number of reaching definitions, distance from last definition or use of a particular data item, or the set of future possible uses for a data item defined at a given statement would help make maintenance efforts easier.

New methods for modeling data dependencies are being developed. Bieman and Debnath [49] have defined the Generalized Program Graph which reflects both data dependency and control flow in the same model. Vouk and Tai [50] have been studying the effect of control flow on bounds on data dependencies. Weiser [51] has developed a method for isolating the control flow that produces data dependencies for specific variables in a program. All the information provided by these and similar studies can be combined with the methods presented in the previous chapters to derive data dependency properties and measures. These measures would complement the control flow measures to provide a more comprehensive view of a program.

The topic of expression structure, unlike control flow and data flow, has been largely ignored. In some existing measures, when expression complexity values are needed, developers either simply count tokens or rely on Halstead's Software Science measures [25] to provide accurate quantifications. However, like programs, expressions also have a control structure which can be expressed in evaluation tree form. From this expression tree, properties can be derived that reflect precedence and other factors of evaluation. Although these properties would help little in understanding arithmetic expressions, which are usually coded verbatim from specifications, they would be especially useful for clarifying boolean expressions,

which are derived less directly from the specifications. Measures to quantify expression properties would be useful as an integral part of a guideline for coding conditional expressions, especially if it is shown that certain expression structures are difficult to comprehend. Again, a rigorous characterization of expression structure will help make empirical evaluations more reliable than existing validation techniques.

Additional areas that will benefit from the methods presented here include defining properties of data structures, concurrency, abstractions, and recursion. These methods do not have to be restricted to imperative programs. Their application to examining properties of functional programming styles provides another topic for future research.

The intuitive appeal of the measures, combined with their rigorous definition and analytical validation demonstrates that basic analytical techniques can be employed to make measures research a true engineering discipline. What is now considered by some to be an art form based largely on intuition can be transformed to a discipline worthy of the term "software engineering".

VII. BIBLIOGRAPHY

1. A. Melton and D. A. Gustafson. "A Model for Software Complexity Measures and Their Uses and A Proposed Method for Classifying Software Complexity Measures." Tech. Rep., Dept. Computer Science, Kansas State University, Manhattan, KS, 1985.
2. T. J. McCabe. "A Complexity Measure." *IEEE Trans. Software Engineering*, SE-2, 4 (1976), 308-320.
3. T. Gilb. *Software Metrics*. Cambridge, MA: Winthrop Publishers, 1977.
4. E. T. Chen. "Program Complexity and Programmer Productivity." *IEEE Trans. Software Engineering*, SE-4, 3 (1978), 187-194.
5. B. A. Sheil. "The Psychological Study of Programming." *ACM Computing Surveys*, 13, 1 (1981), 101-120.
6. W. A. Harrison and K. I. Magel. "A Complexity Measure Based on Nesting Level." *ACM Sigplan Notices*, 16, 3 (1981), 63-74.
7. L. Weissman. "Psychological Complexity of Computer Programs: An Experimental Methodology." *ACM Sigplan Notices*, 9, 6 (1974), 25-36.
8. A. R. Feuer and E. B. Fowlkes. "Relating Computer Program Maintainability to Software Measures." *Proc. National Computer Conference*, Montvale, NJ: AFIPS Press, (1979), 1003-12.
9. G. M. Weinberg, D. P. Geller, and T. W-S. Plum. "IF-THEN-ELSE Considered Harmful." *ACM Sigplan Notices*, 10, 8 (1975), 34-44.
10. D. E. Knuth. "Structured Programming with GO TO Statements." *ACM Computing Surveys*, 6, 4 (1974), 261-301.
11. W. A. Wulf. "Programming Without the Goto." *Information Processing 71*, (1972), 408-413.
12. M. Evangelist. "An Analysis of Control Flow Complexity." *Proc. COMPSAC 84*, Silver Spring, MD: IEEE Computer Society Press, (1984), 388-396.
13. M. S. Hecht. *Flow Analysis of Computer Programs*. New York: Elsevier, 1977.
14. W. A. Harrison, K. I. Magel, R. Kluczny, and A. DeKock. "Applying Software Complexity Metrics to Program Maintenance." *Computer*, 15, 9 (1982), 65-79.
15. S. B. Sheppard, B. Curtis, P. Milliman, and T. Love. "Modern Coding Practices and Programmer Performance." *Computer*, 12, 12 (1979), 41-49.
16. N. Schneidewind and H.-M. Hoffman. "An Experiment in Software Error Data Collection and Analysis." *IEEE Trans. Software Engineering*, SE-5, 3 (1979), 276-286.
17. G. J. Myers. "An Extension to the Cyclomatic Measure of Program Complexity." *ACM Sigplan Notices*, 12, 10 (1977), 61-64.
18. W. J. Hansen. "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)." *ACM Sigplan Notices*, 13, 3 (1978), 29-33.
19. A. L. Baker and S. H. Zweben. "A Comparison of Measures of Control Flow Complexity." *IEEE Trans. Software Engineering*, SE-6, 6 (1980), 506-512.
20. A. L. Baker. "Software Science and Program Complexity Measures." Ph.D. Dissertation, The Ohio State University, Columbus, OH, 1979.

21. V. R. Basili. "Product Metrics." In *Tutorial on Models and Metrics for Software Management and Engineering*, Ed. V. R. Basili. Silver Spring, MD: IEEE Computer Society Press, 1980, pp. 214-217.

22. H. E. Dunsmore and J. D. Gannon. "Analysis of the Effect of Programming Factors on Programming Effort." *Journal of Systems and Software*, 1 (1980), 141-153.

23. V. R. Basili and A. J. Turner. "A Transportable and Extendible Compiler." *Software-Practice and Experience*, 5, 3 (1975), 269-278.

24. W. A. Harrison and K. I. Magel. "A Topological Analysis of the Complexity of Computer Programs with Less Than Three Binary Branches." *ACM Sigplan Notices*, 16, 4 (1981), 51-63.

25. M. H. Halstead. *Elements of Software Science*. New York: Elsevier, 1977.

26. P. Piwowarski. "A Nesting Level Complexity Measure." *ACM Sigplan Notices*, 17, 9 (1982), 44-50.

27. M. R. Woodward, M. A. Hennell, and D. Hedley. "A Measure of Control Flow Complexity in Program Text." *IEEE Trans. Software Engineering*, SE-5, 1 (1979), 45-50.

28. M. H. Williams. "Generating Structured Flow Diagrams: The Nature of Unstructuredness." *The Computer Journal*, 20, 1 (1977), 45-50.

29. J. M. Bieman. "Measuring Software Data Dependency Complexity." Ph. D. Dissertation, University of Louisiana, Lafayette, LA, 1984.

30. R. D. Gordon. "Measuring Improvements in Program Clarity." *IEEE Trans. Software Engineering*, SE-5, 2 (1979), 79-90.

31. J. Lassez, D. Van Der Knijff, and J. Sheppard. "A Critical Examination of Software Science." *Journal of Systems and Software*, 2 (1981), 105-112.

32. J. L. Elshoff. "An Investigation into the Effects of the Counting Method Used on Software Science Measurements." *ACM Sigplan Notices*, 13, 2 (1978), 30-45.

33. V. Y. Shen, S. D. Conte, and H. E. Dunsmore. "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support." *IEEE Trans. Software Engineering*, SE-9, 2 (1983), 155-165.

34. J. L. Elshoff. "An Analysis of Some Commercial PL/I Programs." *IEEE Trans. Software Engineering*, SE-2, 2 (1976), 113-120.

35. E. Oviedo. "Control Flow, Data Flow and Program Complexity." *Proc. COMPSAC 80*, Silver Spring, MD: IEEE Computer Society Press, (1980), 146-152.

36. N. Chapin. "A Measure of Software Complexity." *Proc. National Computer Conference*, Montvale, NJ: AFIPS Press, (1979), 995-1002.

37. S. Henry and D. Kafura. "Software Structure Metrics Based on Information Flow." *IEEE Trans. Software Engineering*, SE-7, 5 (1981), 510-518.

38. M. H. Whitworth and P. A. Szulewski. "The Measurement of Control and Data Flow Complexity in Software Designs." *Proc. COMPSAC 81*, Silver Spring, MD: IEEE Computer Society Press, (1981), 735-743.

39. E. W. Dijkstra. "Goto Statement Considered Harmful." *Communications of the ACM*, 11, 3 (1968), 147-148.

40. M. R. Paige. "Program Graphs, an Algebra, and Their Implication for Programming." *IEEE Trans. Software Engineering*, SE-1, 3 (1975), 286-291.

41. M. R. Paige. "An Analytical Approach to Program Testing." *Proc. COMPSAC 78*, Silver Spring, MD: IEEE Computer Society Press, (1980), 527-531.

42. A. L. Baker, J. W. Howatt, and J. M. Bieman. "Criteria for Finite Sets of Paths that Characterize Control Flow." Tech. Rep. 85-19, Dept. Computer Science, Iowa State University, Ames, IA, 1985.

43. A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Reading, MA: Addison-Wesley, 1977.

44. T. Lengauer and R. E. Tarjan. "A Fast Algorithm for Finding Dominators." *ACM Transactions on Programming Languages and Systems*, 1, 1 (1979), 121-141.

45. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Reading, MA: Addison-Wesley, 1983.

46. A. N. Gileadi and H. F. Ledgard. "On a Proposed Measure of Program Structure." *ACM Sigplan Notices*, 9, 5 (1974), 31-36.

47. R. Kosaraju. "Analysis of Structured Programs." *Journal of Computing and System Science*, 9, 3 (1974), 232-255.

48. H. F. Ledgard and M. Marcotty. "A Genealogy of Control Structures." *Communications of the ACM*, 18, 11 (1975), 629-639.

49. J. M. Bieman and N. C. Debnath. "An Analysis of Software Structure Using a Generalized Program Graph." *Proc. COMPSAC85*, Silver Spring, MD: IEEE Computer Society Press, (1985), 254-259.

50. M. A. Vouk and K. C. Tai. "Sensitivity of Definition-Use Data Flow Metrics to Control Structures." Department of Computer Science Technical Report, North Carolina State University, Raleigh, NC, 1985.

51. M. Weiser. "Program Slicing." *IEEE Trans. Software Engineering*, SE-10, 4 (1984), 352-357.

VIII. ACKNOWLEDGEMENTS

First, and foremost, I thank Al Baker for motivating this research, spending many fruitful hours discussing its development, and showing me the forest when I got lost in the trees. This work is certainly as much his as it is mine. As important was the support of my family. I could not have succeeded as easily without their understanding, especially in these last few months. Jim Bieman, Rob Bugh and Narayan Debnath contributed needed criticism and ideas that helped make the work more complete. Deb Knox provided constructive editorial criticism and advice; I am sure this thesis reads better because of her help. As much appreciated as the academic and moral support is the wherewithal provided by the Air Force Institute of Technology, support that enabled me to attend school in the first place.

IX. APPENDIX: NESTING EVALUATION TOOL FLOWGRAPHS

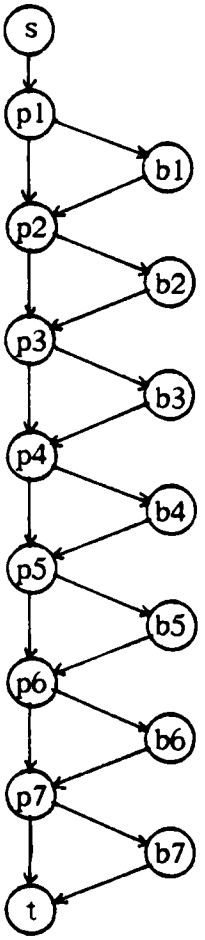


Figure IX-1. Flowgraph SA with n = 7 predicates

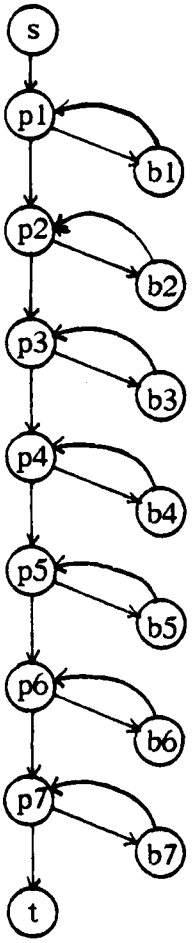


Figure IX-2. Flowgraph SI with n = 7 predicates

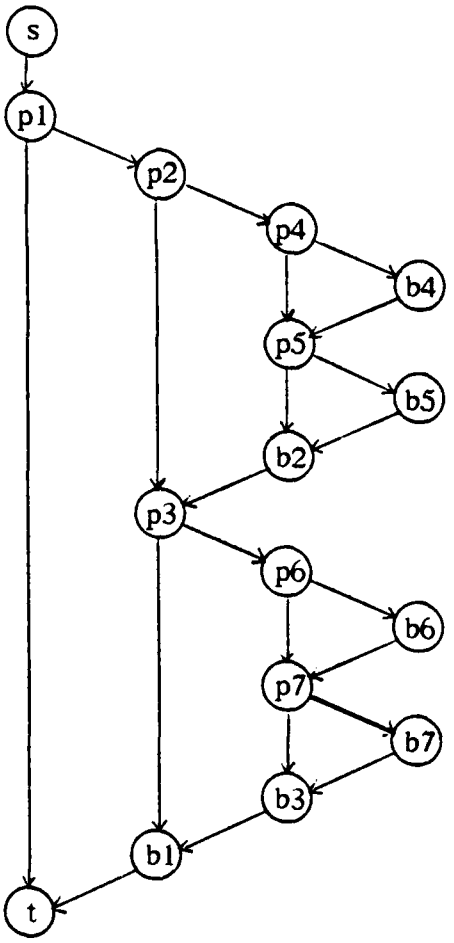


Figure IX-3. Flowgraph INA with n = 7 predicates

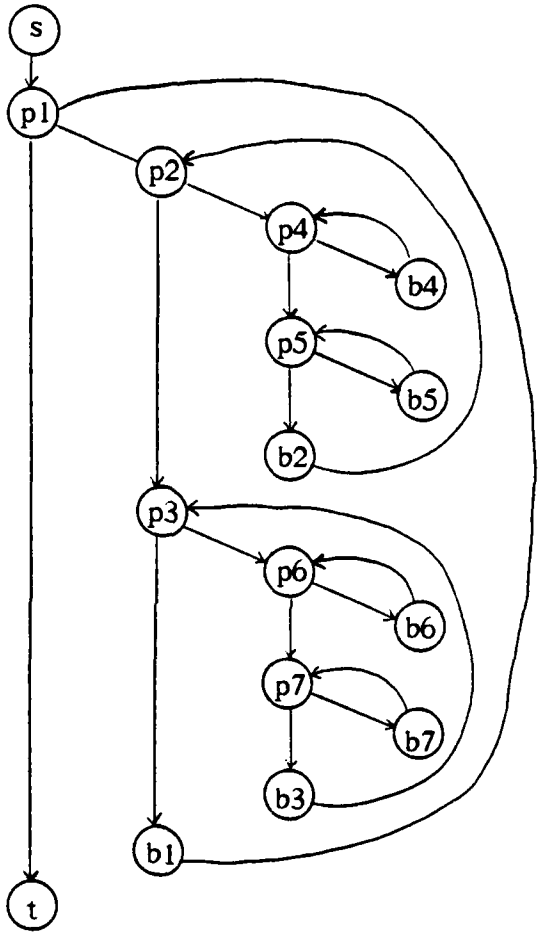


Figure IX-4. Flowgraph INI with n = 7 predicates

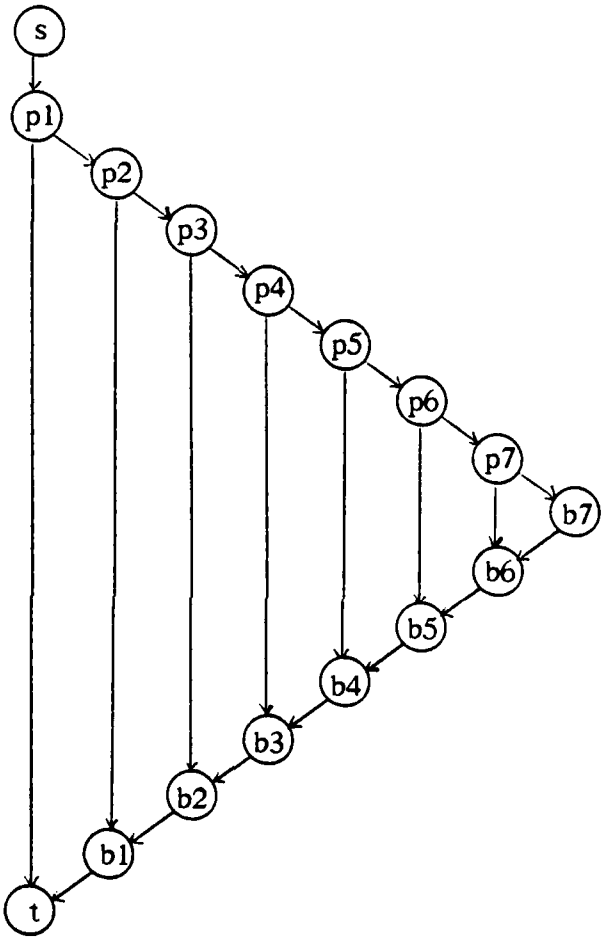


Figure IX-5. Flowgraph MNA with n = 7 predicates

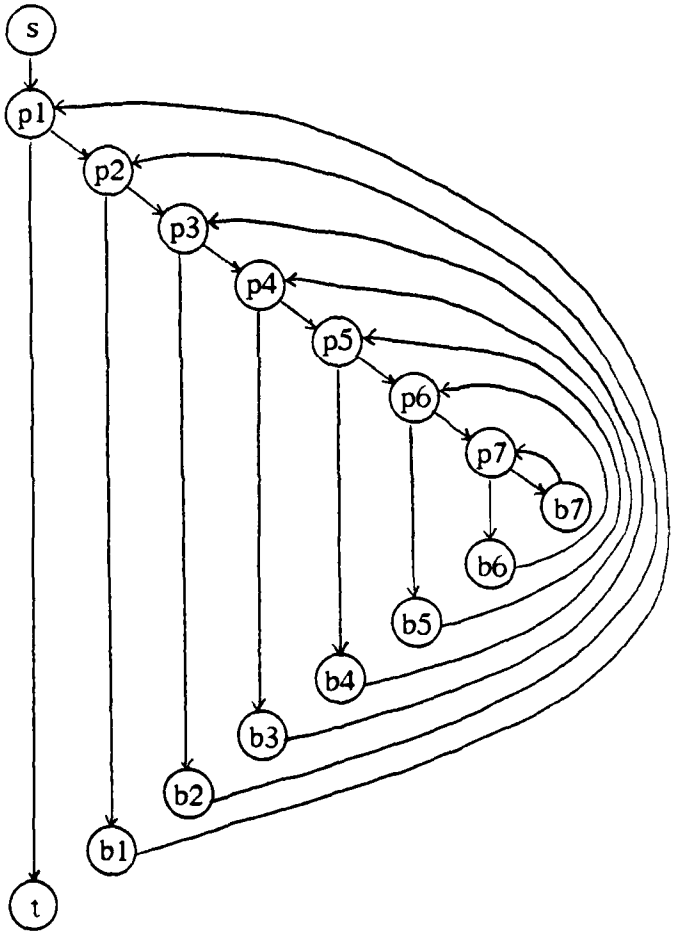


Figure IX-6. Flowgraph MNI with n = 7 predicates

END

DTIC

5-86